Incremental Relational Lenses

Incremental Relational Lenses

RUDI HORN, University of Edinburgh, United Kingdom
ROLY PERERA, University of Edinburgh and University of Glasgow, United Kingdom
JAMES CHENEY, University of Edinburgh, United Kingdom

Lenses are a popular approach to bidirectional transformations, a generalisation of the *view update* problem in databases, in which we wish to make changes to *source* tables to effect a desired change on a *view*. However, perhaps surprisingly, lenses have seldom actually been used to implement updatable views in databases. Bohannon, Pierce and Vaughan proposed an approach to updatable views called *relational lenses*, but to the best of our knowledge this proposal has not been implemented or evaluated to date. We propose *incremental relational lenses*, that equip relational lenses with change-propagating semantics that map small changes to the view to (potentially) small changes to the source tables. We also present a language-integrated implementation of relational lenses and a detailed experimental evaluation, showing orders of magnitude improvement over the non-incremental approach. Our work shows that relational lenses can be used to support expressive and efficient view updates at the language level, without relying on updatable view support from the underlying database.

CCS Concepts: • **Software and its engineering → Domain specific languages**; • **Information systems → Database views**; *Relational database query languages*;

Additional Key Words and Phrases: lenses, bidirectional transformations, relational calculus, incremental computation

ACM Reference Format:
Rudi Horn, Roly Perera, and James Cheney. 2018. Incremental Relational Lenses. *Proc. ACM Program. Lang.* 2, ICFP, Article 74 (September 2018), 30 pages. https://doi.org/10.1145/3236769

1 INTRODUCTION

A typical web application is based on a three-tier architecture with user interaction on the client (web browser), application logic on the server, and data stored in a (typically relational) database. Relational databases employ SQL query and update expressions, including *projections*, *selections* and *joins*, which correspond closely to familiar list comprehension operations in functional languages. Relational databases offer data persistence and high performance and are flexible enough for a range of applications. However, the *impedance mismatch* between database queries and conventional programming makes even simple programming tasks challenging [Copeland and Maier 1984]. Languages such as C# [Meijer et al. 2006], F# [Syme 2006; Cheney et al. 2013], Links [Cooper et al. 2006], and Ur/Web [Chlipala 2015], and libraries such as Database-Supported Haskell [Ulrich and Grust 2015], have partly overcome this challenge using *language-integrated query*, in which query expressions are integrated into the host language and type system.

Authors' addresses: Rudi Horn, University of Edinburgh, United Kingdom, r.horn@ed.ac.uk; Roly Perera, University of Edinburgh and University of Glasgow, United Kingdom, roly.perera@ed.ac.uk; James Cheney, University of Edinburgh, United Kingdom, jcheney@inf.ed.ac.uk.

2475-1421/2018/9-ART74
https://doi.org/10.1145/3236769

However, language support for database programming is still incomplete. For example, *views* are a fundamental concept in databases [Ramakrishnan and Gehrke 2003] that are typically not supported in language-integrated query. A view is a relation defined over the database tables using a relational query. Views have many applications:

(1) A *materialised* view can *precompute* query answers, avoiding expensive recomputation;
(2) a *security* view shows just the information a user needs, while hiding confidential data;
(3) finally, views can be used to define the *external schema* of a database, presenting the data in a form convenient for a particular application, or for *integrating* data from different databases.

Most databases allow specifying views using a variant of table creation syntax, and views can then be used in much the same way as regular database tables. It is therefore natural to wish to *update* a view; for example, making a security view updatable would make sense if the user is intended to have write access to the view data but should not have write access to the underlying table. Unfortunately, it is nontrivial to update views. Some view updates may correspond to multiple possible updates to the source tables, while others may not be translatable at all [Dayal and Bernstein 1982; Bancilhon and Spyratos 1981]. Most relational database systems only allow selection and projection operations in updatable views, so updating views defined using joins is not allowed.

Lenses were introduced by Foster et al. [2007] as a generalisation of updatable views to arbitrary data structures. A lens is a pair of functions $get : S \to V$ and $put : S \times V \to S$, subject to the following *round-tripping* or *well-behavedness* laws:

$$get(put(s, v)) = v \qquad put(s, get(s)) = s$$

Lenses are particularly well-suited to programming tasks where it is necessary to maintain consistency between 'the same' data stored in different places, as often arises in web programming. A great deal of research on *bidirectional programming* has considered this problem, especially in the functional programming community [Stevens 2007; Bohannon et al. 2006a; Hidaka et al. 2010; Foster et al. 2007, 2010; Diskin et al. 2011; Hofmann et al. 2011, 2012; Wang et al. 2011].

Perhaps surprisingly, relatively little of this work has considered view updates in databases. The most important exception is the work of Bohannon et al. [2006a], who proposed lens combinators for projections, selections and joins on relational data, and proved their well-behavedness. These *relational lenses* are defined using *put* functions which map the source database and an updated view to the updated source database. Bohannon et al.'s work showed that it is possible in principle for databases to support updatable views including joins, provided the type system tracks *integrity constraints* on the data, such as functional dependencies.

However, to the best of our knowledge, the practicality of relational lenses has not been demonstrated. The proposed definitions of *get* and *put* are *state-based* — they showed how to compute the view from the base table state, and how to compute the entire new state of the base tables from the updated view and the old table state. These definitions suggest an obvious, if naive, implementation strategy: computing the new source table contents and replacing the old contents. This is simply impractical for any realistic database, and is usually hugely wasteful, in the common case when updates affect relatively few records. Replacing source tables would also necessitate locking access to the affected tables for long periods, destroying any hope of concurrent access.

Luckily, replacing entire source tables with their new contents is seldom necessary. The reason is that updates to tables (and views) are often small: for example, a row might be inserted or deleted, or a single field value modified. Indeed, there is a large literature on the problem of *incremental view maintenance* [Gupta and Mumick 1995; Koch 2010; Koch et al. 2016] addressing the problem of how to modify a materialised view to keep it consistent with changes to the source tables. The benefits of incremental evaluation are not confined to databases, either: witness the growing literature on

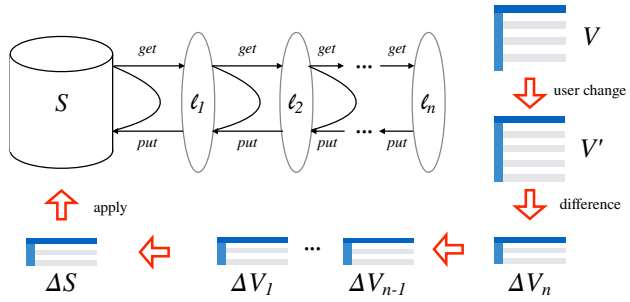


Fig. 1. Propagating changes through lenses from view to source.

adaptive and *incremental* functional programming [Acar et al. 2006; Hammer et al. 2014; Cai et al. 2014]. Indeed, the foundations of change-oriented bidirectional transformations have even been investigated in the form of *edit lenses* [Hofmann et al. 2012] and other formalisms. It is natural to ask whether incrementalisation can be used to make relational lenses practical.

In this paper, we propose *incremental relational lenses*. Figure 1 illustrates our approach. Given a lens ℓ (defined by composing several primitive lenses ℓ_i), and the initial view value $V = get_\ell(S)$, suppose V is updated to V'. We begin by calculating a *view delta* (i.e. change set) $\Delta V = V' \ominus V$. Here $\ominus$ is the operation that calculates a delta mapping one value to another. Then, for each step ℓ_i of the definition of ℓ, we translate the view delta ΔV_i of V_i to a source delta ΔV_{i-1}. We do this by defining an incremental version of the *put* operation, δput, which takes S and ΔV as arguments, and which satisfies the following law:

$$put(S, get(S) \oplus \Delta V) = S \oplus \delta put(S, \Delta V)$$

where $\oplus$ denotes the application of a delta to a value. Finally, once we have calculated the source delta $\Delta S = \Delta V_0$, we translate it to a sequence of SQL `INSERT`, `UPDATE` and `DELETE` commands.

Our approach avoids recomputing and replacing entire tables. Moreover, it can often translate small view deltas to small source deltas. Working with small deltas reduces the amount of computation and data movement incurred. On the other hand, incremental relational lenses still may need to access the source tables to compute correct deltas. We show that this can be done efficiently by issuing auxiliary queries during delta propagation.

We have implemented incremental relational lenses in Links [Cooper et al. 2006], a web programming language with comprehensive support for language-integrated queries. Our experiments show that incremental evaluation offers dramatic performance benefits over the naive state-based approach, just as one would hope or even expect. Perhaps more importantly, we prove the correctness of our approach. The state-based relational lens definitions have a number of subtleties, and proving the correctness of their incremental versions is a nontrivial challenge. Since relational lenses use set-based rather than multiset-based semantics, recent work by Koch [2010] and Cai et al. [2014] on incrementalising multiset operations does not apply; instead, we build on classical work on incremental view updates [Qian and Wiederhold 1991; Griffin et al. 1997]. Incremental relational lenses are also related to edit lenses [Hofmann et al. 2012] and some other frameworks; we discuss the relationship in detail in Section 7.

Outline and contributions. In the rest of this paper, we present necessary background on relational lenses and incrementalisation, define and prove the correctness of incremental relational lenses, and empirically validate our implementation to establish practicality.

- § 2 shows how relational lenses are integrated into Links, and illustrates our incremental semantics for relational lenses via examples.
- § 3 presents background from Bohannon et al. [2006a], including auxiliary concepts and their state-based definitions of relational lenses.
- § 4 introduces our framework for incremental relational queries.
- § 5 presents our main contribution, incremental relational lenses, along with proofs of correctness of optimised forms of their δput operations.
- § 6 presents our experimental results.
- §§ 7 and 8 discuss related work and present our conclusions.

2 RELATIONAL LENSES BY EXAMPLE

In this section we illustrate the use of relational lenses as integrated into Links, a web and database programming language [Cooper et al. 2006]. We first illustrate the naive implementation of relational lenses and then show the incremental approach. We use an example from Bohannon et al. [2006a] involving a small database of albums and tracks, and an updatable view that can be defined over it using relational lenses. In Links, it is straightforward to define a web-based user interface for editing such a view; we elide those details. We also suppress the technical details of relational lenses and incrementalisation until later sections.

In Links, we can initialise a database connection using the **`database`**[1] expression, e.g.:

```
var db = database "music_database";
```

We can then define table references that correspond to actual database tables, using the `table` expression; for example, we can define references to the tables "albums" and "tracks" as follows:

```
var albumsTable = table "albums"
                  with (album: String, quantity: Int)
                  tablekeys [["album"]] from db;

var tracksTable = table "tracks"
                  with (track: String, date: Int, rating: Int, album: String)
                  tablekeys [["track", "album"]] from db;
```

In a **`table`** expression, the table name and names of fields and their types are given, and the **`tablekeys`** clause provides a list of lists of field names. Each element of this list should be a *key* in the sense that the combined field values are unique in the table; for example, in `albumsTable` there is at most one row with a given `album` field, whereas in `tracksTable` the track and album fields together uniquely identify a row. Key constraints are a special kind of *functional dependency*, discussed further below.

Links supports language integrated queries and updates over base tables. We have extended Links with updatable views based on relational lenses. Relational lens definitions require the specification of *functional dependencies* for base tables, constraints that indicate which attributes determine the value of other attributes. As an example, a functional dependency $album \rightarrow quantity$ says that if two rows have the same `album` attribute, they must also have the same `quantity` value.

The **`lens`** keyword makes a base table into a basic lens, and allows specifying additional functional dependencies beyond key constraints [2]. We define basic lenses for each of our tables: the *albums* table has a functional dependency $album \rightarrow quantity$, and the *tracks* table has a functional dependency $track \rightarrow date\ rating$, which says that *date* and *rating* depend on *track*. This implies that any

[1] Information such as the hostname, port, username and password needed to access the database also needs to be provided; this can be included in the **`database`** expression or, as here, in a separate configuration file.

[2] Such constraints are not currently checked in our implementation, but this is straightforward and an orthogonal concern.

track may appear in different albums, but should have the same *date* and *rating*. (These functional dependencies are as specified by Bohannon et al. [2006a], but result in a database with some redundancy; however, we keep the example as is for ease of comparison with prior work.)

```
var albumsLens = lens albumsTable with album -> quantity;
var tracksLens = lens tracksTable with track -> date rating;
```

A common workflow within a web application is to extract some view of the data from the database, associate it with a form, and map form responses to updated versions of the associated data. If the mapping from the database to the form data is defined by a lens, then the view can be fetched from the lens using the *get* operation. When the user submits the form, the *put* operation of the lens should allow us to propagate the changes to the underlying database. So, we can add a row to *albumsTable* as follows:

```
var albums = get albumsLens;
var newAlbums = albums ++ [(album='Disintegration', quantity=1)]
put albumsLens with newAlbums;
```

Because `albumsLens` is simply a basic lens wrapping `albumsTable`, this is equivalent to just using a SQL-style update, written in Links as follows:

```
insert into albumsTable values [(album='Disintegration', quantity=1)]
```

In practice, views are more useful for selecting subsets of data or combining tables. For example, the web application might show a form allowing updates to a single album at a time, such as `'Galore'`. The updatable form data can be extracted using a select lens on *tracksLens* which requires rows to have *album* = `'Galore'`. We then call *get* on the select lens, make any desired changes and then use *put* on the select lens and the new view to update the database.

```
/* create a lens which selects only the tracks from 'Galore' */
var selectLens = lensselect from tracksLens where album = 'Galore';
/* get the smaller subset of tracks */
var tracks = get selectLens;
/* ... newTracks = updates to tracks ... */
/* update database */
put selectLens with newTracks;
```

Suppose `tracksTable` contains the entries specified in Figure 2 on the left. Calling *get* on the lens produces the view on the right, containing only the records having *album* = `'Galore'`. If the user changes the rating of the track `'Lullaby'` to 4, then submits the form, an updated view is generated as shown on the right in Figure 3. The application can then call *put* on the view, which will cause the underlying *tracksTable* table to be updated with the changes to the `'Lullaby'` tracks. Notice that we must change both tracks because of the functional dependency $track \rightarrow date\ rating$. This will produce the updated table as shown on the left in Figure 3.

Type and Integrity Constraints. Both views and base tables can be associated with integrity constraints, and updated views need to respect these constraints. There are three kinds of constraints:

(1) The updated view should be well-typed in the usual sense. Views that have rows with extra or missing fields, or field values of the wrong types, are ruled out statically by the type system.
(2) The updated view should satisfy the functional dependencies associated with the view. Thus, the functional dependency $track \rightarrow date\ rating$ from our example implies that we cannot change the rating or date of `'Lullaby'` in one row without changing all the others to match.
(3) The updated view may also need to satisfy a predicate on the rows. Views defined by lenses may have selection conditions, such as *album* = `'Galore'`. Inserting rows with other *album* values, or changing this field in existing rows, is not allowed.

track	date	rating	album
'Lullaby'	1989	3	'Galore'
'Lullaby'	1989	3	'Show'
'Lovesong'	1989	5	'Galore'
'Lovesong'	1989	5	'Paris'
'Trust'	1992	4	'Wish'

$\overset{get}{\Longrightarrow}$

track	date	rating	album
'Lullaby'	1989	3	'Galore'
'Lovesong'	1989	5	'Galore'

Fig. 2. Select lens example: computing the view (right) from the source (left) using *get*

track	date	rating	album
'Lullaby'	1989	**4**	'Galore'
'Lullaby'	1989	**4**	'Show'
'Lovesong'	1989	5	'Galore'
'Lovesong'	1989	5	'Paris'
'Trust'	1992	4	'Wish'

$\overset{put}{\Longleftarrow}$

track	date	rating	album
'Lullaby'	1989	**4**	'Galore'
'Lovesong'	1989	5	'Galore'

Fig. 3. Select lens example: computing the new source (left) using *put* on the new view (right) and old source (from Fig. 2, left). The change to the view results in two changes to the source.

These constraints all originate in the definition of *schemas* for relational lenses introduced by Bohannon et al. [2006a]. The correctness properties of relational lenses rely on these integrity constraints, and if the updated view satisfies its constraints then the updated underlying table will also satisfy its own constraints.

Incremental View Updates. Bohannon et al. [2006a] define the *get* and *put* directions of relational lenses as set-theoretic expressions showing how to compute the new source given the old source and the updated view. The most obvious approach to implementing the *put* behavior of a relational lens is to use these definitions to calculate the new source table 'from scratch' and replace the old one with the new one. For example, in Figure 3 this would mean deleting all five tuples of the old tracks table and then re-inserting the three unchanged tuples and the new versions of the two modified ones. We could accomplish the desired effect in Figure 3 using SQL UPDATE operations to change just the ratings of the `'Lullaby'` tracks to 4. This would typically be more efficient (especially if there were many more unaffected rows).

Therefore, we adopt an incremental approach, as outlined in the introduction. Instead of working with the entire tables, we first compute a delta for the modified view, that is, sets of rows to be inserted and deleted. We illustrate deltas as tables with rows annotated with '+' (for insertion) or '-' (for deletion). An example delta for the update shown in Figure 3 is shown on the left of Figure 4. This delta is then used to calculate a delta for the source table, as shown on the right side of Figure 4.

Once we have computed the change set for the underlying tables from Figure 4, we can use the delta and other available information (such as **`tablekeys`** declarations) to produce SQL update commands that perform the desired update. For our example, we can perform the needed updates using two SQL `update` operations, as follows:

```
update tracks set date = 1989 rating = 4 where track = 'Lullaby' and album = 'Galore';
update tracks set date = 1989 rating = 4 where track = 'Lullaby' and album = 'Show';
```

	track	date	rating	album
−	'Lullaby'	1989	3	'Galore'
+	'Lullaby'	1989	4	'Galore'
−	'Lullaby'	1989	3	'Show'
+	'Lullaby'	1989	4	'Show'

$\delta put \Leftarrow$

	track	date	rating	album
−	'Lullaby'	1989	3	'Galore'
+	'Lullaby'	1989	4	'Galore'

Fig. 4. Select lens example: using δput, we compute the source delta (left-hand side) from the view delta (right-hand side) and the original source (Fig. 2, left).

Composition. So far we have discussed only one relational lens primitive, namely selection. Updatable views can also be defined using relational lenses for dropping attributes (projection) or combining data from several tables (joining). We can combine these primitive relational lenses using the general definition of composition for lenses [Foster et al. 2007].

The join operation raises a subtle issue: joining a table with itself can lead to copying data, which lacks clean bidirectional semantics [Foster et al. 2007]. Bohannon et al. [2006a] implicitly used a linear type discipline for relational lenses, forbidding repeated use of the same base table in different parts of a view. We do not currently check this constraint in Links, but there is no fundamental obstacle to doing so, and our formalisation also enforces it.

We extend our track example as shown in Figure 5 by first joining the two tables. This gives us a view `joinLens` containing all tracks and their corresponding albums and album quantities. We may then decide to discard the *date* attribute using a projection lens, yielding view `dropLens`. (The **`lensdrop`** combinator includes a **`default`** value giving a value to use when new data is inserted into the view.) Finally we use selection to define a view `selectLens` retaining rows with quantity greater than 2. Figure 6 shows each of the lenses in blue, and along the left shows how the composite lens's *get* produces the table in the bottom left with the three tracks `'Lullaby'`, `'Lovesong'` and `'Trust'`. We show the intermediate views in the *get* direction for completeness, but it is not necessary to compute them explicitly; we can compose the *get* directions and extract a single SQL query to produce the final output. The query for the example in Figure 5 would be:

```
select t1.track, t1.rating, t1.album, t2.quantity
   from tracks as t1
   join albums as t2 on t1.album = t2.album
   where t2.quantity > 2;
```

Suppose a user then makes the changes shown in red at the bottom of Figure 6. Performing the update with composed lenses works similarly to the case for single lenses: for a composite lens $\ell_1; \ell_2$ we first propagate the view delta backwards through ℓ_2 to obtain a source delta, then treat that as a view delta for ℓ_1. We calculate an initial delta by comparing the updated view with the original view for the last lens. This is shown at the bottom of Figure 6: comparing the original view with the updated table yields the change set shown at the bottom right.

All intermediate change sets are calculated using the previous change set and by querying the database. Since the (non-incremental) *put* function is defined in terms of the previous source and updated view, sometimes we need to know parts of the values of the old source or old view to calculate the incremental behaviour. Therefore, for some relational lens steps we need to run one or more queries against the database during change propagation. The select lens is an example: in order to ensure that the source update preserves the functional dependency $track \rightarrow date\ rating$, we need to query the database to find out what other album/track rows might need to have their ratings updated. The drop lens step also illustrates the need for auxiliary querying, in this case to

```
var joinLens = lensjoin albumsLens with tracksLens on album;
var dropLens = lensdrop date determined by track default 2018 from joinLens;
var selectLens = lensselect from dropLens where quantity > 2;
```

Fig. 5. A view `selectLens` defined by composing join, drop and select operators.

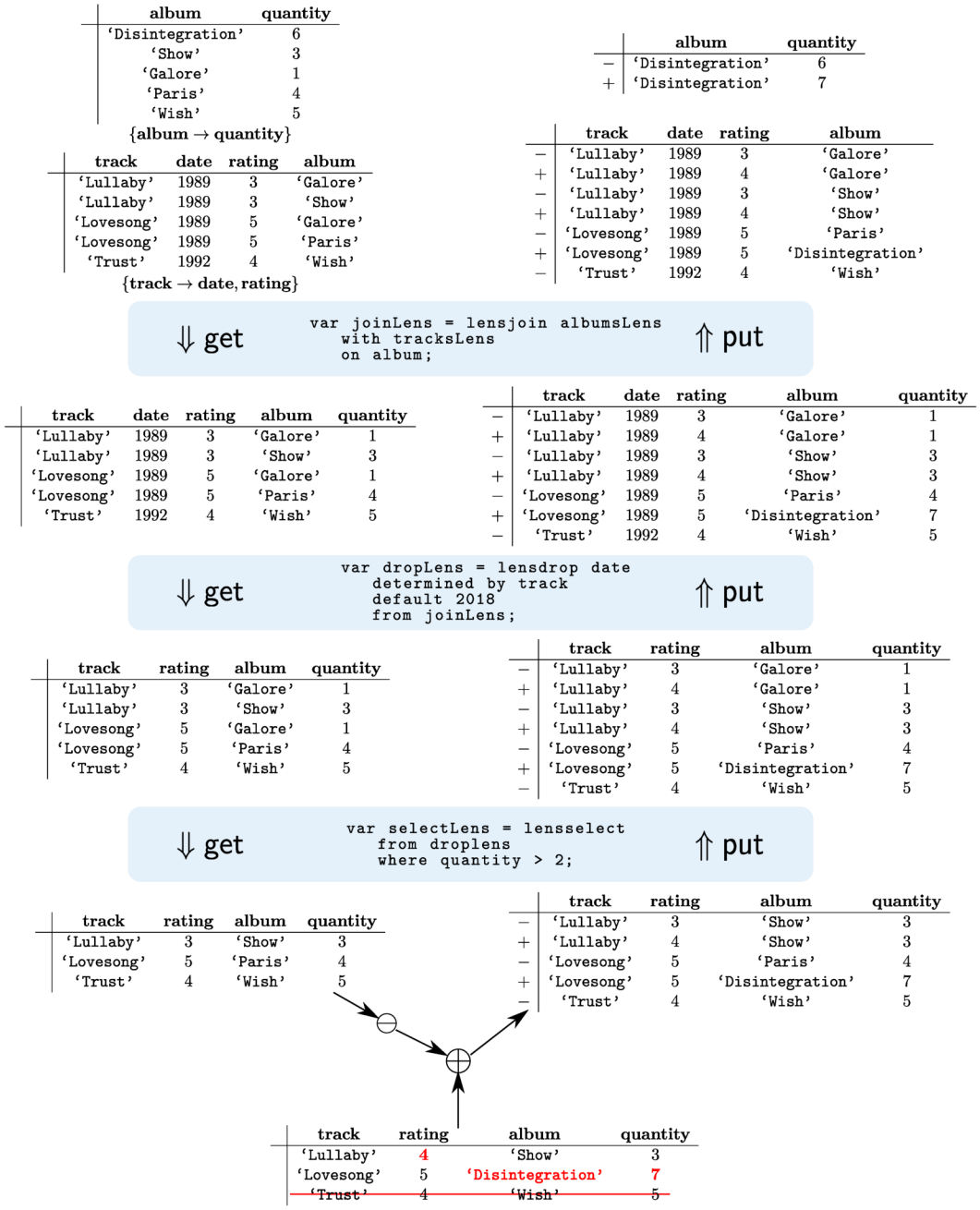


Fig. 6. An example of how an update propagates through `selectLens`. Changes to the view are shown at the bottom in red.

Queries	$q, q' ::=$	$M \mid R \mid \texttt{let}\ R = q\ \texttt{in}\ q'$	Relations, names and let binding
	$\mid$	$q - q' \mid q \cup q' \mid q \cap q'$	Set operations
	$\mid$	$\sigma_P(q) \mid \pi_U(q) \mid q \bowtie q' \mid \rho_{A/B}(q)$	Relational algebra
Predicates	$P, Q ::=$	$\top \mid \neg P \mid P \wedge Q \mid P \vee Q$	Logical connectives
	$\mid$	$A = B \mid A = a \mid X \in q$	Tuple predicates
	$\mid$	$\pi_U(P) \mid P \bowtie Q \mid \rho_{A/B}(q)$	Relational algebra

Fig. 7. Syntax of relational expressions and predicates

find out the dropped dates of rows that are being updated. Finally, the join lens splits the changes of the joined view into changes for the individual tables; this too may require querying the underlying data. This produces the deltas shown in the top right corner of Figure 6.

Finally, we convert the source deltas into SQL update commands to update the underlying tables. Again we make use of table key information to generate concise updates, as follows:

```
update albums set quantity = 7 where album = 'Disintegration';
update tracks set date = 1989 rating = 4 where track = 'Lullaby' and album = 'Galore';
update tracks set date = 1989 rating = 4 where track = 'Lullaby' and album = 'Show';
delete from tracks where track = 'Lovesong' and album = 'Paris';
insert into tracks (track, date, rating, album)
values ('Lovesong', 1989, 5, 'Disintegration');
delete from tracks where track = 'Trust' and album = 'Wish';
```

3 STATE-BASED RELATIONAL LENSES

In this section we recapitulate background concepts from database theory [Abiteboul et al. 1995] and then review the definitions of relational lenses [Bohannon et al. 2006a]. We use different notation from that paper in some cases, and explain the differences as necessary.

3.1 Database Preliminaries

3.1.1 Attributes and Records. *Attribute names*, or simply *attributes*, are ranged over by A, B, C and attribute values by a, b, c. Records m, n are partial functions from attributes to attribute values. For simplicity, we assume a single (unwritten) type for attribute values; of course, in our implementation we support the usual integers, strings, booleans, etc. Records are written $\{A = a, B = b, \ldots\}$. Identifiers U, V range over sets of attributes considered as record domains; we use X, Y, Z for arbitrary sets. We write $m : U$ to indicate that $dom(m) = U$. Basic operations on records include:

- record projection $m[V]$, which means record m domain-restricted to $dom(m) \cap V$;
- domain antirestriction $m\backslash_V$, which means m domain-restricted to $dom(m) - V$;
- record update $m \leftarrow\!\!+\, n : U \cup V$ where $m : U, n : V$, which defines $(m \leftarrow\!\!+\, n)(A)$ as $n(A)$ if $A \in V$ and $m(A)$ otherwise.

Given attribute $A \in U$ and $B \notin U$, we write $U[A/B]$ for $(U - \{A\}) \cup \{B\}$, and similarly if $m : U$ then $m[A/B] : U[A/B]$ is the tuple resulting from renaming attribute A in M to B. Renaming is definable as $(m\backslash_{\{A\}}) \leftarrow\!\!+\, \{B = m(A)\}$.

3.1.2 Relations. *Relations* M, N, O are (finite) sets of records with the same domain. M has domain U, or equivalently M is a relation of type U, written $M : U$, if $m : U$ for all $m \in M$. Relations are closed under the standard operations of relational algebra; Figure 7 defines the syntax of relational expressions q. This includes relation constants M, relation names R, and a `let` construct

we include for convenience. The operations $-$, $\cup$ and $\cap$ have their usual set-theoretic interpretation, subject to the constraint that the arguments q, q' have the same type U, i.e. $q, q' : U$. We explain the remaining relational algebra operations in this section.

Relational projection is record projection extended to relations:

$$\pi_U(M) \stackrel{\text{def}}{=} \{m[U] \mid m \in M\}$$

Given $M : U$ and $N : V$, their *natural join* is defined by

$$M \bowtie N = \{m : U \cup V \mid m[U] \in M \text{ and } m[V] \in N\}$$

P ranges over predicates, which we can interpret as (possibly infinite) sets of records, or equivalently as functions from records to Booleans. Predicates are required for specifying selection filters in relational selection, as well as for the specification of filter conditions in lens definitions. We write $P : U$ to indicate a predicate over records with domain U. The predicate $A = B$ holds for records m satisfying $m(A) = m(B)$, while $A = a$ holds when $m(A) = a$, and $X \in q$ holds when $m[X]$ is in the result of query q. The predicates $\top$ (truth), $\neg P$ (negation), and $P \wedge Q$ (conjunction) are interpreted as usual. For convenience we include predicates $\pi_U(P)$, $P \bowtie Q$, and $\rho_{A/B}(P)$ which behave analogously to the relational operations, if we view predicates as sets of records. For example, $\pi_U(P)$ holds for records u such that $t[U] = u$ for some t satisfying P.

Given a predicate P and relation M, the selection $\sigma_P(M)$ is defined as follows:

$$\sigma_P(M) \stackrel{\text{def}}{=} \{m \in M \mid P(m)\}$$

We will be interested in cases where predicates are insensitive to the values of certain attributes; we write "P ignores U" when $P(m)$ can be determined without considering any of the values that m assigns to attributes in U — i.e. when for all m and n, if $m\backslash_U = n\backslash_U$ then $P(m) \iff P(n)$.

We define the relational renaming operation $\rho_{A/B}(M)$ as

$$\rho_{A/B}(M) \stackrel{\text{def}}{=} \{m[A/B] \mid m \in M\}$$

which makes it possible to join tables with differing column names. As mentioned above, we also write $\rho_{A/B}(P)$ for the result of renaming attribute A in predicate P to B.

3.1.3 Functional Dependencies. A *functional dependency* is a pair of sets of attributes, written $X \to Y$. We say $X \to Y$ is a functional dependency *over* U, written $X \to Y : U$, iff $X \cup Y \subseteq U$. If $X \to Y$ is a functional dependency over U and $M : U$, then M *satisfies* $X \to Y$, written $M \vDash X \to Y$, iff $m[X] = n[X]$ implies $m[Y] = n[Y]$ for all $m, n \in M$. We write $m, M \vDash X \to Y$ as a shorthand for $\{m\} \cup M \vDash X \to Y$. It is conventional in database theory to write sets of attributes such as $\{A, B, C\}$ as $A\ B\ C$, and $A \to B\ C$ to mean the functional dependency $\{A\} \to \{B, C\}$.

Typically we work with sets F, G of functional dependencies over a fixed U and write $F : U$ iff $X \to Y : U$ for every $X \to Y \in F$. The notation $M \vDash F$ means that $M \vDash X \to Y$ for all $X \to Y \in F$. Likewise, $F \vDash G$ means that $M \vDash F$ implies $M \vDash G$ for any M, and $F \equiv G$ means that $F \vDash G$ and $G \vDash F$. We write $F[A/B]$ for the result of renaming attribute A to B in all functional dependencies in F, i.e. $F[A/B] \stackrel{\text{def}}{=} \{X[A/B] \to Y[A/B] \mid X \to Y \in F\}$.

3.1.4 Relation Types and Database Schemas. Bohannon et al. [2006a] employ *relation types* $Rel(U, P, F)$, where U is an attribute set, P is a predicate over U, and F is a set of functional dependencies over U. A value M of type $Rel(U, P, F)$ is a relation $M : U$ such that $P(m) = \top$ for each $m \in M$ and $M \models F$. (Bohannon et al. [2006a] wrote (U, P, F) instead of $Rel(U, P, F)$; we prefer the more descriptive notation.)

Bohannon et al. [2006a] also defined lenses between *relational schemas* Σ mapping relation names to relation types $Rel(U, P, F)$, and relational lens combinators referred to the relations in the source

and target schemas by their names. We adopt an alternative, but equivalent approach. Schemas have the following syntax:

$$\Sigma ::= Rel(U, P, F) \mid \Sigma \otimes \Sigma'$$

Thus a database schema is a product of one or more relation types, written $Rel(U_1, P_1, F_1) \otimes \cdots \otimes Rel(U_n, P_n, F_n)$. Values of schema type $\Sigma \otimes \Sigma'$ are pairs of values of type Σ and Σ'. We use tensor product notation $\otimes$ instead of $\times$ to indicate that schema types obey a linear typing discipline. When we work with relational lenses, the initial schema will consist of the tensor product of all tables that contribute to the view we wish to define; we assume that a table appears at most once in a value of such a schema.

3.2 Lenses

A *lens* [Foster et al. 2007] $\ell : S \Leftrightarrow V$ is a bidirectional transformation between two sets S and V, where S is a set of source values and V is a set of possible views, determined by two functions, get_ℓ and put_ℓ, with the following signatures:

$$get_\ell : S \to V \qquad\qquad put_\ell : S \times V \to S$$

A lens is *well-behaved* if it satisfies two round-tripping properties relating get_ℓ and put_ℓ. The property PutGet ensures that whatever data we put into a lens is returned unchanged if we get it again. The property GetPut ensures that if we put the view value back into the lens unchanged, the underlying source value is also unchanged.

$$get_\ell(put_\ell(s, v)) = v \tag{PutGet}$$
$$put_\ell(s, get_\ell(s)) = s \tag{GetPut}$$

From now on the notation $\ell : S \Leftrightarrow V$ means that ℓ is a well-behaved lens from S to V.

Lenses form a category with identity and composition constructions. The identity lens $id_X : X \Leftrightarrow X$ is given by the functions *get* and *put* defined as:

$$get_{id}(x) = x \qquad\qquad put_{id}(x, x') = x'$$

We omit the subscript on *id* when clear from context. The identity lens is trivially well-behaved. Diagram-order composition $\ell_1; \ell_2 : X \Leftrightarrow Z$ of the lenses $\ell_1 : X \Leftrightarrow Y$ and $\ell_2 : Y \Leftrightarrow Z$ is given by the functions *get* and *put* defined as:

$$get_{\ell_1;\ell_2}(x) = get_{\ell_2}(get_{\ell_1}(x)) \qquad put_{\ell_1;\ell_2}(x, z) = put_{\ell_1}(x, put_{\ell_2}(get_{\ell_1}(x), z))$$

As discussed in previous work [Hofmann et al. 2011, 2012], the category of lenses also has symmetric monoidal products; that is, there is a construction $\otimes$ on its objects such that $X \otimes Y$ is the set of pairs $\{(x, y) \mid x \in X, y \in Y\}$, and which satisfies symmetry and associativity laws:

$$X \otimes Y \equiv Y \otimes X \tag{Sym}$$
$$X \otimes (Y \otimes Z) \equiv (X \otimes Y) \otimes Z \tag{Assoc}$$

These laws are witnessed by (invertible) lenses $sym_{X,Y}$ and $assoc_{X,Y,Z}$, defined as follows:

$$get_{sym}(x, y) = (y, x) \qquad get_{assoc}(x, (y, z)) = ((x, y), z)$$
$$put_{sym}(\_, (y, x)) = (x, y) \qquad put_{assoc}(\_, ((x, y), z)) = (x, (y, z))$$

In addition, we have the following combinator for combining two lenses 'side-by-side':

$$get_{\ell_1 \otimes \ell_2}(x_1, x_2) = (get_{\ell_1}(x_1), get_{\ell_2}(x_2))$$
$$put_{\ell_1 \otimes \ell_2}((x_1, x_2), (y_1, y_2)) = (put_{\ell_1}(x_1, y_1), put_{\ell_2}(x_2, y_2))$$

so that if $\ell_1 : X_1 \Leftrightarrow Y_1$ and $\ell_2 : X_2 \Leftrightarrow Y_2$ then $\ell_1 \otimes \ell_2 : X_1 \otimes X_2 \Leftrightarrow Y_1 \otimes Y_2$.

3.3 Relational Revision

A key relational lens concept introduced by Bohannon et al. [2006a] is *relational revision*. Given a set of functional dependencies $F : U$ and relations $M, N : U$ such that $N \models F$, relational revision modifies M to M' so that $M' \cup N \models F$. For example, given $F = \{A \to B\}$ and $M = \{\{A = 1, B = 2\}, \{A = 2, B = 3\}\}$ and $N = \{\{A = 1, B = 42\}\}$, the result of revising M to be consistent with N and F is $\{\{A = 1, B = 42\}, \{A = 2, B = 3\}\}$.

3.3.1 Functional Dependencies in Tree Form. In general revision may not be uniquely defined, for example if there are cycles among functional dependencies. Bohannon et al. [2006a] avoid this problem by requiring that sets of functional dependencies be in a special form called *tree form*. We briefly restate the definition for concreteness.

Definition 3.1. Given functional dependencies F, define

$$V_F = \{X \mid X \to Y \in F\} \cup \{Y \mid X \to Y \in F\} \qquad E_F = \{(X, Y) \mid X \to Y \in F\}$$

Then we say F is in *tree form* if the graph $T_F = (V_F, E_F)$ is a forest and V_F partitions $\bigcup V_F$.

If F is in tree form, then each attribute set of F corresponds to a node in a tree (or forest) where the edges correspond to elements of F. Moreover, no distinct nodes of T_F have common atttibutes. For example, $\{A \to B\,C, B \to D\}$ is not in tree form, but is equivalent to $\{A \to B, A \to C, B \to D\}$ which is in tree form. However, $\{A \to B\,C, C \to A\,D\}$ has no equivalent tree form representation.

Figure 8 defines various functions on sets of functional dependencies. The sets $\mathit{left}(F)$ and $\mathit{right}(F)$ consist of all attributes appearing on the left or right side of any functional dependency $X \to Y \in F$. The set $\mathit{outputs}(F)$ consists of all attributes that are actually constrained in F by other attributes. Finally, $\mathit{roots}(F)$ is the set of all nodes of T_F that have indegree zero. From now on, we assume sets of functional dependencies F appearing in relation types $\mathit{Rel}(U, P, F)$ are in tree form, as a prerequisite to this type being well-formed.

3.3.2 Revision and Merge Operations. Relational revision is expressed in terms of a *record revision* operation $\mathit{recrevise}_F(m, N)$ which takes a set of functional dependencies $F : U$ in tree form, a record $m : U$, and a set of records $N : U$ such that $N \models F$, and is defined by recursion over the tree structure of F. If F is empty, record revision simply returns m. Otherwise, there must be at least one functional dependency $X \to Y$ in F such that X is a root. If m and some $n \in N$ have the same values for X, we return $m \leftarrow n[Y]$, that is, a copy of m whose Y attributes have been updated with those from $n[Y]$; otherwise we return m unchanged. We then recursively process the remaining functional dependencies.

Traversing F starting from the roots is nondeterministic, but provided F is in tree form, the end result of record revision is uniquely defined, because each attribute in $\mathit{right}(F)$ is modified at most once and no attribute can be modified until all other attributes it depends on have been modified.

Definition 3.2 (Relational revision). Figure 8 defines the *relational revision* operation $\mathit{revise}_F(M, N)$ that takes two sets of records $M : U$ and $N : U$ where $N \vDash F$, and applies record revision to every record $m \in M$ using the given functional dependencies F.

Definition 3.3 (Relational merge). Figure 8 also defines the *relational merge* operation $\mathit{merge}_F(M, N)$, where $N \vDash F$, which revises M according to F and N and then unions the result with N.

3.4 Relational Lens Primitives

In this section we recapitulate the primitive relational lenses introduced by Bohannon et al. [2006a]: a *selection* lens that corresponds to selection, a *drop* lens that corresponds to projection, and a *join* lens that corresponds to relational join. ("Corresponds" means that the *get* direction coincides with

$$
\begin{aligned}
\mathit{left}(F) &= \bigcup \{X \mid X \to Y \in F\} \\
\mathit{right}(F) &= \bigcup \{Y \mid X \to Y \in F\} \\
\mathit{outputs}(F) &= \bigcup \{Y \mid \exists X.F \vDash X \to Y \text{ and } X \cap Y = \varnothing\} \\
\mathit{roots}(F) &= \{X \mid \exists Y.X \to Y \in F \text{ and } X \cap \mathit{right}(F) = \varnothing\}
\end{aligned}
$$

$$
F = \{X \to Y\} \cdot F' \iff F = \{X \to Y\} \uplus F' \text{ and } X \in \mathit{roots}(F)
$$

$$
\begin{aligned}
\mathit{recrevise}_F &: U \to \mathit{Rel}(U, F, P) \to U \\
\mathit{recrevise}_\varnothing(m, N) &= m \\
\mathit{recrevise}_{\{X \to Y\} \cdot F}(m, N) &= \begin{cases} \mathit{recrevise}_F(m \leftrightarrow n[Y], N) & \text{if } \exists n \in N.m[X] = n[X] \\ \mathit{recrevise}_F(m, N) & \text{otherwise} \end{cases} \\
\mathit{revise}_F(M, N) &= \{\mathit{recrevise}_F(m, N) \mid m \in M\} \\
\mathit{merge}_F(M, N) &= \mathit{revise}_F(M, N) \cup N
\end{aligned}
$$

Fig. 8. Operations on functional dependencies and relational revision

$$
\begin{aligned}
\text{Relational lenses} \quad \ell, \ell' ::= {} & \texttt{select}_P \mid \texttt{drop}\ A\ \texttt{determined by}\ (X, a) \mid \texttt{join\_dl} \mid \texttt{rename}_{A/B} \\
\mid {} & \texttt{id} \mid \ell_1; \ell_2 \mid \texttt{sym} \mid \texttt{assoc} \mid \ell_1 \otimes \ell_2
\end{aligned}
$$

Fig. 9. Syntax of relational lens expressions

the relational operation.) We also introduce a trivial *rename* lens corresponding to the relational renaming operator. The syntax of the relational lenses, including the generic operations from § 3.2, is given in Figure 9.

Our presentation differs from that of Bohannon et al. [2006a] in that we use generic lens combinators arising from the symmetric monoidal product structure to deal with linearity. This makes it possible for each primitive to mention only the affected source and target data and not the rest of the database instance.

Relational lenses are lenses between schema types Σ, that is, tensor products of relation types $\mathit{Rel}(U, P, F)$. Relational lens expressions are subject to a typing judgement given in Figure 10; well-typed lenses are guaranteed to be well-behaved. The preconditions in these rules are those given by Bohannon et al. [2006a], to which we refer the reader for further explanation.

3.4.1 Select Lens. The lens $\texttt{select}_P : \mathit{Rel}(U, Q, F) \Leftrightarrow \mathit{Rel}(U, P \wedge Q, F)$ is defined as follows by the functions *get* and *put*:

$$
\begin{aligned}
\mathit{get}(M) &= \sigma_P(M) \\
\mathit{put}(M, N) &= \texttt{let}\ M_0 = \mathit{merge}_F(\sigma_{\neg P}(M), N)\ \texttt{in} \\
&\quad\ \texttt{let}\ N_\# = \sigma_P(M_0) - N\ \texttt{in} \\
&\quad\ M_0 - N_\#
\end{aligned}
$$

The *put* operation first calculates M_0, the set of records $\sigma_{\neg P}(M)$ excluded from the original view, revised to be consistent with the functional dependencies witnessed by the updated view N together with N itself. The set $N_\#$ collects records matching P but not in N, which are removed from M_0 in order to satisfy PutGet.

$$\frac{Q \text{ ignores } \mathit{outputs}(F)}{\texttt{select}_P : \mathit{Rel}(U, Q, F) \Leftrightarrow \mathit{Rel}(U, P \wedge Q, F)} \text{ T-Select}$$

$$\frac{P = \pi_{U-A}(P) \bowtie \pi_A(P) \qquad \{A = a\} \in \pi_A(P)}{\texttt{drop } A \texttt{ determined by } (X, a) : \mathit{Rel}(U, P, F \uplus \{X \to A\}) \Leftrightarrow \mathit{Rel}(U - A, \pi_{U-A}(P), F)} \text{ T-Drop}$$

$$\frac{G \vDash U \cap V \to V \qquad P \text{ ignores } \mathit{outputs}(F) \qquad Q \text{ ignores } \mathit{outputs}(G)}{\texttt{join\_dl} : \mathit{Rel}(U, P, F) \otimes \mathit{Rel}(V, Q, G) \Leftrightarrow \mathit{Rel}(U \cup V, P \bowtie Q, F \cup G)} \text{ T-JoinDL}$$

$$\frac{A \in U \qquad B \notin U}{\texttt{rename}_{A/B} : \mathit{Rel}(U, P, F) \Leftrightarrow \mathit{Rel}(U[A/B], \rho_{A/B}(P), F[A/B])} \text{ T-Rename}$$

Fig. 10. Typing rules for relational lens primitives

3.4.2 Project Lens. The lens $\texttt{drop } A \texttt{ determined by } (X, a) : \mathit{Rel}(U, P, F \uplus \{X \to A\}) \Leftrightarrow \mathit{Rel}(U - A, \pi_{U-A}(P), F)$ is defined as follows [3] by the functions *get* and *put*:

$$\begin{array}{rcl} \mathit{get}(M) & = & \pi_{U-A}(M) \\ \mathit{put}(M, N) & = & \texttt{let } M' = N \bowtie \{\{A = a\}\} \texttt{ in} \\ & & \mathit{revise}_{X \to A}(M', M) \end{array}$$

For *put*, each row in N is initially given the default value a for A. M is then used to override the default value in M' using relational revision, so that if there is an entry $m \in M'$ with the same value for the determining column X the corresponding A value from M is used instead.

3.4.3 Join Lens. Bohannon et al. [2006a] described several variants of lenses for join operations. All three perform the natural join of their two input relations in the *get* direction, but differ in how deletions are handled in the *put* direction. A view tuple deletion could be translated to a deletion in the left, right, or both source relations, and so there are three combinators `join_dl`, `join_dr`, and `join_both` expressing these three alternatives. In their extended report, Bohannon et al. [2006b] showed how to define all three combinators as a special case of a generic template. To keep the presentation simple, we present just the `join_dl` combinator here.

The 'join/delete left' lens $\texttt{join\_dl} : \mathit{Rel}(U, P, F) \otimes \mathit{Rel}(V, Q, G) \Leftrightarrow \mathit{Rel}(U \cup V, P \bowtie Q, F \cup G)$ is given by the functions *get* and *put* defined as follows:

$$\begin{array}{rcl} \mathit{get}(M, N) & = & M \bowtie N \\ \mathit{put}((M, N), O) & = & \texttt{let } M_0 = \mathit{merge}_F(M, \pi_U(O)) \texttt{ in} \\ & & \texttt{let } N' = \mathit{merge}_G(N, \pi_V(O)) \texttt{ in} \\ & & \texttt{let } L = (M_0 \bowtie N') - O \texttt{ in} \\ & & \texttt{let } M' = M_0 - \pi_U(L) \texttt{ in} \\ & & (M', N') \end{array}$$

The intuition for the *put* direction is as follows. We first compute M_0 by merging the projection $\pi_U(O)$ into source table M, and likewise N', merging the projection $\pi_V(O)$ into the source table N. We next identify those tuples L which are in the join of M_0 and N' but which are not present in the updated view O. To satisfy PutGet, we must make sure these tuples do not appear in the join after

[3] This is slightly simpler than, but equivalent to, the definition given by Bohannon et al. [2006a]. The proof that the two definitions are equivalent is provided in the full version of the paper [Horn et al. 2018].

updating the source relations. It is sufficient to delete each of those records from one of the two source tables; since this lens deletes uniformly from the left table, we compute M' by subtracting the projection $\pi_U(L)$ from M_0. Finally, M' and N' are the new values for the source tables.

3.4.4 Renaming Lens. Renaming is a trivial but important operation in relational algebra, since otherwise there is no way to join the A field of one table with the $B \neq A$ field of another. We introduce a renaming lens $\texttt{rename}_{A/B} : Rel(U, P, F) \Leftrightarrow Rel(U[A/B], \rho_{A/B}(P), F[A/B])$, provided $A \in U$ and $B \notin U$, with its *get* and *put* operations defined as follows:

$$\begin{array}{rcl} get(M) & = & \rho_{A/B}(M) \\ put(\_, N) & = & \rho_{B/A}(N) \end{array}$$

Bohannon et al. [2006a] did not define such a lens, but its well-behavedness is obvious.

4 INCREMENTAL FRAMEWORK

To describe the incremental behaviour of relational lenses, we need to represent changes to query results in a simple, compositional way. We adopt an approach similar to Griffin et al. [1997], who model "delta relations" as disjoint pairs of relations specifying tuples to be added and removed from a relation of the same type. We use notation similar to Cai et al. [2014] and our relations and delta relations form a *change structure* in their sense, though we do not spell out the details formally.

4.1 Delta Relations

Definition 4.1 (Delta relation). A *delta relation over* U is a pair $\Delta M = (\Delta M^+, \Delta M^-)$ of disjoint relations $\Delta M^+ : U$ and $\Delta M^- : U$. The empty delta relation $(\varnothing, \varnothing)$ is written $\varnothing$. We write $\Delta M : \Delta U$ to indicate that ΔM is a delta relation over U.

A delta specifies a modification to a relation: for example, if $M = \{2, 3, 4\}$ and $\Delta M = (\{3, 5\}, \{4, 9\})$ then ΔM^+ specifies that $\{3, 5\}$ are to be added to M, resulting in the set $\{1, 3, 5\}$. Note that the redundant insertion of 3 specified by ΔM^+ and the redundant deletion of 9 specified by ΔM^- are both permitted. However, Griffin et al. [1997] define a delta $\Delta M : \Delta U$ to be *minimal* for $M : U$ if it contains no redundant insertions or deletions of that sort; for example, $(\{5\}, \{4\})$ is the minimal delta relative to M equivalent to ΔM above.

Definition 4.2 (Minimal delta). $\Delta M : \Delta U$ is *minimal* for $M : U$ iff $\Delta M^+ \cap M = \varnothing$ and $\Delta M^- \subseteq M$.

Definition 4.3 (Implicit coercion to delta-relation). Any relation $M : U$ can be implicitly coerced to a delta-relation $M : \Delta U \stackrel{\text{def}}{=} (M, \varnothing)$ which is minimal for $\varnothing : U$.

Deltas of the same type can be combined by a minimality-preserving *merge* operation $\oplus$.

Definition 4.4 (Delta merge). For any $\Delta M, \Delta N : \Delta U$, define

$$(\Delta M \oplus \Delta N) : \Delta U \stackrel{\text{def}}{=} ((\Delta M^+ - \Delta N^-) \cup (\Delta N^+ - \Delta M^-), (\Delta M^- - \Delta N^+) \cup (\Delta N^- - \Delta M^+))$$

Implicit coercion of M to the delta-relation $(M, \varnothing)$, combined with delta merge $\oplus$, gives rise to a notion of *delta application* $M \oplus \Delta M$. If ΔM is minimal then the resulting delta has an empty negative component and can be coerced back to a relation.

Lemma 4.5. *If ΔM is minimal for M then $M \oplus \Delta M = (M - \Delta M^-) \cup \Delta M^+ = (M \cup \Delta M^+) - \Delta M^-$.*

Definition 4.6 (Delta negate). For any $\Delta M : \Delta U$, define $\ominus \Delta M : \Delta U \stackrel{\text{def}}{=} (\Delta M^-, \Delta M^+)$.

Definition 4.7 (Delta difference). For any $\Delta M, \Delta N : \Delta U$, define $(M \ominus N) : \Delta U \stackrel{\text{def}}{=} M \oplus (\ominus N)$.

The implicit coercion to delta-relations gives rise to a notion of relational difference $(M \ominus N) : \Delta U$, not to be confused with $(M - N) : U$, which is the set difference and only removes elements N from M. $M \ominus N$ can be used, for example, to calculate the difference between two views, such that $N \oplus (M \ominus N) = M$.

Lemma 4.8. *Suppose $M : U$ and $N : U$. Then $(M \ominus N) : \Delta U = (M - N, N - M)$. Moreover $M \ominus N$ is minimal for N.*

The following are some useful straightforward properties of deltas:

Lemma 4.9. *Suppose ΔM minimal for M. Then $(M \oplus \Delta M) - M = \Delta M^+$.*

Lemma 4.10. *Suppose ΔM minimal for M. Then $(M \cap (M \oplus \Delta M)) \ominus M = \ominus \Delta M^-$, hence $M - (M \oplus \Delta M) = \Delta M^-$.*

Corollary 4.11. *If ΔM minimal for M then $(M \oplus \Delta M) \ominus M = \Delta M$.*

Proof. By the previous two lemmas, $(M \oplus \Delta M) \ominus M = ((M \oplus \Delta M) - M, M - (M \oplus \Delta M)) = (\Delta M^+, \Delta M^-) = \Delta M$. □

Corollary 4.12. *If ΔM and $\Delta M'$ are minimal for M and $M \oplus \Delta M = M \oplus \Delta M'$ then $\Delta M = \Delta M'$.*

Proof. By Corollary 4.11, $\Delta M = (M \oplus \Delta M) \ominus M = (M \oplus \Delta M') \ominus M = \Delta M'$. □

The property $(M \oplus \Delta M) \ominus M = \Delta M$ is mentioned in Cai et al. [2014] but not required by their definition of change structures. It is very helpful in our setting because it implies that query expressions incrementalise in a unique, compositional way, as we show next.

4.2 Delta-Relational Operations

We now consider how to incrementalise relational operations. For each relational operator, such as $\sigma_P(M)$ or $merge_F(M, N)$, we would like to define an operation that translates deltas to the arguments to a delta to the result. Incremental operations with symbolic names are written with a dot, for example $\dot{\sigma}_P(M)$, while alphabetic names have their incremental counterpart written with a preceding δ, for example $\delta merge_F(M, N)$.

The notion of *delta-correctness* characterises when a function δop with a suitable signature which operates on deltas can be considered to be a valid "incrementalisation" of a non-incremental operation op. As observed by Griffin et al. [1997], composing incremental relational operations is easier if they also preserve minimality, so we build this property into our definition.

Definition 4.13 (Delta-correctness). For any operation $op : X \to Y$, a delta operation $\delta op : X \times \Delta X \to \Delta Y$ is *delta-correct* for op if for any Δx minimal for x, we have:

(1) $\delta op(x, \Delta x)$ is minimal for $op(x)$.
(2) $op(x \oplus \Delta x) = op(x) \oplus \delta op(x, \Delta x)$

We generalise the above definition to binary operations in the obvious way. Delta-correct operations are uniquely determined by the minimality condition:

Lemma 4.14. *If δop is delta-correct then $\delta op(x, \Delta x) = op(x \oplus \Delta x) \ominus op(x)$ provided Δx is minimal for x. In particular, $\delta op(x, \varnothing) = \varnothing$.*

Proof. By lemma 4.8, $op(x \oplus \Delta x) \ominus op(x)$ is minimal for $op(x)$, and by the definition of delta-correctness and lemma 4.8 we have

$$op(x) \oplus \delta op(x, \Delta x) = op(x \oplus \Delta x) = op(x) \oplus (op(x \oplus \Delta x) \ominus op(x))$$

By Corollary 4.12 we can conclude $\delta op(x, \Delta x) = op(x \oplus \Delta x) \ominus op(x)$. □

For ease of composition, we define $op^\dagger(x, \Delta x) = (op(x), \delta op(x, \Delta x))$ as the function that returns both the updated result and the delta.

Definition 4.15. For any $op : X \longrightarrow Y$, define $op^\dagger : X \times \Delta X \longrightarrow Y \times \Delta Y$ as

$$op^\dagger(x, \Delta x) = (op(x), \delta op(x, \Delta x))$$

We say $op^\dagger$ is delta-correct (with respect to op) when δop is.

If δop is delta-correct then whenever Δx is minimal for x, so is $\delta op(x, \Delta x)$ for $op(x)$. This implies composability in the following sense:

Lemma 4.16. *If $\delta op_1 : X \times \Delta X \longrightarrow \Delta Y, \delta op_2 : Y \times \Delta Y \longrightarrow Z$ are delta-correct then $\delta op_2 \circ {op_1}^\dagger$ and ${op_2}^\dagger \circ {op_1}^\dagger$ are delta-correct (both with respect to $op_2 \circ op_1$).*

Furthermore, this implies we may incrementalise any function built up out of incrementalisable relational operations, by replacing ordinary operators with their incremental counterparts, largely as described by Cai et al. [2014]. Given a query $q(R_1, \ldots, R_n)$, we can transform it to a delta-correct (but not necessarily efficient) incremental version by taking $\delta(q) = \texttt{let}\ (R, \Delta R) = (q)^\dagger\ \texttt{in}\ \Delta R$, where the transformation $(\cdot)^\dagger$ is defined as follows:

$$\begin{array}{rclrcl} (M)^\dagger & = & (M, \varnothing) & (op(q_1, \ldots, q_n))^\dagger & = & (op^\dagger((q_1)^\dagger, \ldots, (q_n)^\dagger)) \\ (R)^\dagger & = & (R, \Delta R) & (\texttt{let}\ R = q\ \texttt{in}\ q')^\dagger & = & \texttt{let}\ (R, \Delta R) = (q)^\dagger\ \texttt{in}\ (q')^\dagger \end{array}$$

Essentially $(q)^\dagger$ traverses the query, replacing relation variables with pairs of variables and deltas, replacing constant relations with pairs $(M, \varnothing)$ and dealing with individual operations and let-bindings compositionally. We abuse notation slightly by adding syntax for pairs.

Theorem 4.17. *If $q : Rel(U_1) \times \cdots \times Rel(U_n) \longrightarrow Rel(U)$ then $\delta(q)$ and $(q)^\dagger$ are delta-correct with respect to q.*

4.3 Optimisation Rules for Delta Operations

To sum up, we have established that for any query there is a an extensionally unique incrementalisation, obtained by computing the difference between the updated query result and the original result. Of course, this is far from an efficient implementation strategy. In this section, we present a number of optimisation rules for incremental relational operations, as well as relational revision and merge.

Most of the following characterisations of incremental relational operations are presented in prior work such as Griffin et al. [1997], but without detailed proofs; we include detailed proofs in the full version [Horn et al. 2018].

Lemma 4.18. *[Valid optimisations] Assume ΔM, ΔN are minimal for M, N respectively. Then:*

(1) $\dot{\sigma}_P(M, \Delta M) = (\sigma_P(\Delta M^+), \sigma_P(\Delta M^-))$
(2) $\dot{\pi}_U(M, \Delta M) = (\pi_U(\Delta M^+) - \pi_U(M), \pi_U(\Delta M^-) - \pi_U(M \oplus \Delta M))$
(3) $(M, \Delta M) \dot{\bowtie} (N, \Delta N) = (((M \oplus \Delta M) \bowtie \Delta N^+) \cup (\Delta M^+ \bowtie (N \oplus \Delta N)), (\Delta M^- \bowtie N) \cup (M \bowtie \Delta N^-))$
(4) $\dot{\rho}_{A/B}(M, \Delta M) = (\rho_{A/B}(\Delta M^+), \rho_{A/B}(\Delta M^-))$
(5) *If $N \subseteq M$ and $N \oplus \Delta N \subseteq M \oplus \Delta M$ then $(M, \Delta M) \dot{-} (N, \Delta N) = \Delta M \ominus \Delta N$*

Relational revision is only used directly for drop lenses, where only the first argument M may change. The following lemma provides an optimisation for this case:

Lemma 4.19. *Suppose $M \models X \longrightarrow A$ and $M \oplus \Delta M \models X \longrightarrow A$. Then $\delta revise_{X \to A}((M, \Delta M), (N, \varnothing)) = (revise_{X \to A}(\Delta M^+, N), revise_{X \to A}(\Delta M^-, N))$.*

For relational merge, the join lens makes use of the following special case:

Lemma 4.20. *If* $merge_F(M, N) = M$ *then* $\delta merge_F((M, \varnothing), (N, \Delta N)) = merge_F(M, \Delta N^+) \ominus M$.

In select and join lenses, we will avoid explicitly recomputing $merge_F(M, N)$ by showing that it is sufficient to consider only a subset of possibly-affected rows in M. We define a function called $affected_F$ which returns a predicate selecting a (hopefully small) superset of the rows that may be changed by relational merge according to F and a set of view records N. The returned predicate is the necessary condition for any changes implied by F and N.

Definition 4.21. $affected_F(N) \stackrel{\text{def}}{=} \bigvee_{X \to Y \in F} X \in \pi_X(N)$.

It is then possible to replace the target relation M with only those rows in M which are likely to be updated, allowing fewer rows to be queried from the database:

Lemma 4.22. *If* $P = affected_F(\Delta N^+)$ *then* $merge_F(M, \Delta N^+) \ominus M = merge_F(\sigma_P(M), \Delta N^+) \ominus \sigma_P(M)$.

5 INCREMENTALISING RELATIONAL LENSES

5.1 Incremental Lenses

Assume S, V are sets of relations equipped with sets of deltas $\Delta S, \Delta V$ and corresponding operations $\oplus, \ominus$. A (well-behaved) *incremental lens* $\ell : S \Leftrightarrow V$ is a well-behaved lens equipped with additional operations $\delta get_\ell : S \times \Delta S \to \Delta V$ and $\delta put_\ell : S \times \Delta V \to \Delta S$ satisfying

$$get_\ell(s \oplus \Delta s) = get_\ell(s) \oplus \delta get_\ell(s, \Delta s)$$
$$put_\ell(s, get_\ell(s) \oplus \Delta v) = s \oplus \delta put_\ell(s, \Delta v) \qquad \text{(}\Delta\text{PutGet)}$$

and such that if Δs is minimal for s then $\delta get_\ell(s, \Delta s)$ is minimal for $get_\ell(s)$, and likewise if Δv is minimal for $get_\ell(s)$ then $\delta put_\ell(s, \Delta v)$ is minimal for s.

The δget_ℓ direction simply performs incremental view maintenance, which is not our main concern here; we include it to show how it fits together with δput_ℓ but do not discuss it further. The first equation and minimality condition is simply delta-correctness of δget_ℓ relative to get_ℓ.

Our focus here is the δput_ℓ operation. In this direction, it would be redundant to supply an argument holding the previous value of the view, since it can be obtained via get_ℓ. The ΔPutGet rule and associated minimality condition is a special case of the delta-correctness rule, where we only consider changes to V, not S:

$$put_\ell(s, get_\ell(s) \oplus \Delta v) = put_\ell(s, get_\ell(s)) \oplus \delta put_\ell(s, \Delta v)$$

and the term $put_\ell(s, get_\ell(s))$ has been simplified to s by the GetPut rule.

We can equip the generic lens combinators from § 3.2 with suitable delta-correct δput operations as follows:

$$\delta put_{\texttt{id}}(\_, \Delta x) = \Delta x$$
$$\delta put_{\texttt{sym}}(\_, (\Delta y, \Delta x)) = (\Delta x, \Delta y)$$
$$\delta put_{\texttt{assoc}}(\_, ((\Delta x, \Delta y), \Delta z)) = (\Delta x, (\Delta y, \Delta z))$$
$$\delta put_{\ell_1;\ell_2}(x, \Delta z) = \delta put_{\ell_1}(x, \delta put_{\ell_2}(get_{\ell_1}(x), \Delta z))$$
$$\delta put_{\ell_1 \otimes \ell_2}((x_1, x_2), (\Delta y_1, \Delta y_2)) = (\delta put_{\ell_1}(x_1, \Delta y_1), \delta put_{\ell_2}(x_2, \Delta y_2))$$

It is straightforward to show that the resulting incremental lenses are well-behaved.

For each relational lens primitive ℓ described in § 3.4, `select`$_P$, `drop` A `determined by` (X, a), `join_dl` and `rename`$_{A/B}$, we will define an incremental δput_ℓ operation as follows. First, we incrementalise the corresponding put_ℓ definition from § 3.4, obtaining a function $\delta Put_\ell : (S \times \Delta S) \times (V \times \Delta V) \to \Delta S$ that is delta-correct with respect to put_ℓ. Since we are only interested in the case where S does not change and $v = get_\ell(s)$, we then specialize this operation to obtain

$\delta put_\ell(s, \Delta v) = \delta Put_\ell((s, \varnothing), (get_\ell(s), \Delta v))$, which yields a well-behaved lens. We then apply further optimisations to simplify this expression to a form that can be evaluated efficiently.

The well-behavedness of the generic lens combinators and the relational lens primitives imply the well-behavedness of any well-typed lens expression.

5.2 Select Lens

The incremental lens $\ell = \delta\texttt{select}_P : Rel(U, Q, F) \Leftrightarrow Rel(U, P \wedge Q, F)$ is the lens $\texttt{select}_P$ of the same type, equipped with δput_ℓ defined as follows:

$$\begin{array}{rcl} \delta put_\ell & : & Rel(U,Q,F) \times \Delta Rel(U, P \wedge Q, F) \to \Delta Rel(U,Q,F) \\ \delta put_\ell(M, \Delta N) & = & \texttt{let } N = \sigma_P(M) \texttt{ in} \\ & & \texttt{let } (M_0, \Delta M_0) = merge^\dagger_F(\sigma^\dagger_{\neg P}(M, \varnothing), (N, \Delta N)) \texttt{ in} \\ & & \texttt{let } (N_\#, \Delta N_\#) = \sigma^\dagger_P(M_0, \Delta M_0) \mathbin{-^\dagger} (N, \Delta N) \texttt{ in} \\ & & (M_0, \Delta M_0) \mathbin{\dot{-}} (N_\#, \Delta N_\#) \end{array}$$

Lemma 5.1. *The incremental select lens $\delta\texttt{select}_P$ is well-behaved.*

Definition 5.2. Define an optimised incremental $\delta\texttt{select}_P$ lens ℓ' with $\delta put_{\ell'}$ defined as follows:

$$\begin{array}{rcl} \delta put_{\ell'}(M, \Delta N) & = & \texttt{let } Q = affected_F(\Delta N^+) \texttt{ in} \\ & & \texttt{let } \Delta M_0 = (merge_F(\sigma_{Q \wedge \neg P}(M), \Delta N^+) \ominus \sigma_{Q \wedge \neg P}(M)) \ominus \Delta N^- \texttt{ in} \\ & & \texttt{let } \Delta N_\# = (\sigma_P(\Delta M_0{}^+), \sigma_P(\Delta M_0{}^-)) \ominus \Delta N \texttt{ in} \\ & & \Delta M_0 \ominus \Delta N_\# \end{array}$$

The optimised version works as follows. ΔM_0 can be calculated by querying the database for $\sigma_{Q \wedge \neg P}(M)$ and then performing relational merge using ΔN^+. The remaining computations involve only deltas and can be performed in-memory. ΔM_0 contains all changes to the underlying table including any removed rows, but does not account for rows which previously didn't satisfy P, but do after the updates. These rows, which would violate lens well-behavedness, are found in $\Delta N_\#$. We calculate $\Delta N_\#$ just using the delta difference operator $\ominus$ because $N_\# \oplus \Delta N_\#$ is always a subset of $M_0 \oplus \Delta M_0$. The final update consists of the changes to the table M_0 merged with the changes to remove all rows in $\Delta N_\#$.

Theorem 5.3. *[Correctness of optimised select lens] Suppose $N = \sigma_P(M)$ where $M : Rel(U, Q, F)$. Suppose also that ΔN is minimal with respect to N and that $N \oplus \Delta N : Rel(U, P \wedge Q, F)$. Then $\delta put_\ell(M, \Delta N) = \delta put_{\ell'}(M, \Delta N)$.*

5.3 Project Lens

The incremental lens $\ell = \delta\texttt{drop } A \texttt{ determined by } (X, a) : Rel(U, P, F) \Leftrightarrow Rel(U - A, \pi_{U-A}(P), F')$, where $F \equiv F' \uplus \{X \to A\}$, is the lens $\texttt{drop } A \texttt{ determined by } (X, a)$ of the same type, equipped with δput_ℓ defined as follows:

$$\begin{array}{rcl} \delta put_\ell & : & Rel(U,P,F) \times \Delta Rel(U - A, \pi_{U-A}(P), F') \to \Delta Rel(U,P,F) \\ \delta put_\ell(M, \Delta N) & = & \texttt{let } N = \pi_{U-A}(M) \texttt{ in} \\ & & \texttt{let } (M', \Delta M') = (N, \Delta N) \bowtie^\dagger (\{\{A = a\}\}, \varnothing) \texttt{ in} \\ & & \delta revise_{X \to A}((M', \Delta M'), (M, \varnothing)) \end{array}$$

Lemma 5.4. *The incremental projection lens $\delta\texttt{drop } A \texttt{ determined by } (X, a)$ is well-behaved.*

Definition 5.5. Define an optimised incremental $\delta\texttt{drop } A \texttt{ determined by } (X, a)$ lens ℓ' with $\delta put_{\ell'}$ defined as follows:

$$\begin{array}{rcl} \delta put_{\ell'}(M, \Delta N) & = & \texttt{let } \Delta M' = (\Delta N^+ \bowtie \{\{A = a\}\}, \Delta N^- \bowtie \{\{A = a\}\}) \texttt{ in} \\ & & (revise_{X \to A}(\Delta M'^+, M), revise_{X \to A}(\Delta M'^-, M)) \end{array}$$

$\Delta M'$ extends $\Delta N'$ with the extra attribute A set to the default value a, to match the domain of the underlying table. The final step optimises the use of $\delta revise_{X \to A}(\cdot,\cdot)$ in δput_ℓ using Lemma 4.19.

Theorem 5.6. *[Correctness of optimised project lens] Suppose* $M : Rel(U,P,F)$ *and* $N = \pi_{U-A}(M)$. *Suppose also that* ΔN *is minimal with respect to* N *and that* $N \oplus \Delta N : Rel(U-A, \pi_{U-A}(P), F')$, *where* $F \equiv F' \uplus \{X \to A\}$. *Then* $\delta put_\ell(M, \Delta N) = \delta put_{\ell'}(M, \Delta N)$.

5.4 Join Lens

The incremental lens $\ell = \delta\texttt{join\_dl} : Rel(U,P,F) \otimes Rel(V,Q,G) \Leftrightarrow Rel(U \cup V, P \bowtie Q, F \cup G)$ is the lens `join_dl` of the same type, equipped with δput_ℓ defined as follows:

$$
\begin{array}{rcl}
\delta put_\ell & : & Rel(U,P,F) \times Rel(V,Q,G) \times \Delta Rel(U \cup V, P \bowtie Q, F \cup G) \\
 & & \to \Delta Rel(U,P,F) \times \Delta Rel(V,Q,G) \\
\delta put_\ell((M,N),\Delta O) & = & \texttt{let } O = M \bowtie N \texttt{ in} \\
 & & \texttt{let } (M_0, \Delta M_0) = merge^\dagger_F((M,\varnothing), \pi^\dagger_U(O,\Delta O)) \texttt{ in} \\
 & & \texttt{let } (N', \Delta N') = merge^\dagger_G((N,\varnothing), \pi^\dagger_V(O,\Delta O)) \texttt{ in} \\
 & & \texttt{let } (L, \Delta L) = ((M_0,\Delta M_0) \bowtie^\dagger (N',\Delta N')) -^\dagger (O,\Delta O) \texttt{ in} \\
 & & \texttt{let } \Delta M' = (M_0,\Delta M_0) \dot{-} \pi^\dagger_U(L,\Delta L) \texttt{ in} \\
 & & (\Delta M', \Delta N')
\end{array}
$$

Lemma 5.7. *The incremental join lens* $\delta\texttt{join\_dl}$ *is well-behaved.*

Definition 5.8. Define an optimised incremental $\delta\texttt{join\_dl}$ lens ℓ' with $\delta put_{\ell'}$ defined as follows:

$$
\begin{array}{rcl}
\delta put_{\ell'}((M,N),\Delta O) & = & \texttt{let } P_M = affected_F(\pi_U(\Delta O^+)) \texttt{ in} \\
 & & \texttt{let } P_N = affected_G(\pi_V(\Delta O^+)) \texttt{ in} \\
 & & \texttt{let } \Delta M_0 = merge_F(\sigma_{P_M}(M), \pi_U(\Delta O^+)) \ominus \sigma_{P_M}(M) \texttt{ in} \\
 & & \texttt{let } \Delta N' = merge_G(\sigma_{P_N}(N), \pi_V(\Delta O^+)) \ominus \sigma_{P_N}(N) \texttt{ in} \\
 & & \texttt{let } \Delta L = (((M \oplus \Delta M_0) \bowtie \Delta N'^+) \cup (\Delta M_0{}^+ \bowtie (N \oplus \Delta N')), \\
 & & \qquad (\Delta M_0{}^- \bowtie N) \cup (M \bowtie \Delta N'^-)) \ominus \Delta O \texttt{ in} \\
 & & \texttt{let } \Delta M' = \Delta M_0 \ominus \pi_U(\Delta L) \texttt{ in} \\
 & & (\Delta M', \Delta N')
\end{array}
$$

In the optimised join lens, ΔM_0 and $\Delta N'$ can be calculated by first querying $\sigma_{P_M}(M)$ and $\sigma_{P_N}(N)$, where P_M and P_N include all rows potentially affected by merging functional dependencies, and then performing the appropriate relational merges using $\pi_U(\Delta O^+)$ and $\pi_V(\Delta O^+)$. Since this step may result in additional rows being generated or deleted rows not being removed, any excess rows ΔL are determined by calculating which rows would have been changed in the joined view after updates to the underlying tables ΔM_0 and $\Delta N'$, and then comparing those to the desired changes ΔO. We can calculate ΔL efficiently by querying the underlying M and N tables only for records having identical join keys to records in ΔM_0 and $\Delta N'$.

Finally, the updated left table can be calculated as the changes to the left table ΔM_0 minus all records that need to be removed to ensure the lens is well behaved. The changes $\Delta N'$ are used for the right table.

Theorem 5.9. *[Correctness of optimised join lens] Suppose* $M : Rel(U,P,F)$ *and* $N : Rel(V,Q,G)$ *and* $O = M \bowtie N$. *Suppose also that* ΔO *is minimal with respect to* O, *and* $O \oplus \Delta O : Rel(U \cup V, P \bowtie Q, F \cup G)$. *Then* $\delta put_\ell((M,N),\Delta O) = \delta put_{\ell'}((M,N),\Delta O)$.

5.5 Rename Lens

The incremental lens $\ell = \delta\,\texttt{rename}_{A/B} : Rel(U,P,F) \Leftrightarrow Rel(U[A/B], \rho_{A/B}(P), F[A/B])$, where $A \in U$ and $B \notin U$, is the lens $\texttt{rename}_{A/B}$ with the additional function δput_ℓ defined as follows:

$$\begin{array}{rcl} \delta put_\ell & : & Rel(U,P,F) \times \Delta Rel(U[A/B], \rho_{A/B}(P), F[A/B]) \to \Delta Rel(U,P,F) \\ \delta put_\ell(M, \Delta N) & = & \texttt{let } N = \rho_{B/A}(M) \texttt{ in} \\ & & \texttt{let } (M', \Delta M') = \rho^{\dagger}_{B/A}(N, \Delta N) \texttt{ in} \\ & & \Delta M' \end{array}$$

Definition 5.10. Define an optimised incremental $\delta\,\texttt{rename}_{A/B}$ lens ℓ' with $\Delta\, put_{\ell'}$ defined as follows:

$$\delta put_{\ell'}(M, \Delta N) \quad = \quad (\rho_{B/A}(\Delta N^+), \rho_{B/A}(\Delta N^-))$$

The optimised rename lens performs the inverse rename operation on both components of the delta relation. No database queries are required.

Theorem 5.11. *[Correctness of rename lens] Suppose $M : Rel(U,P,F)$ and $N = \rho_{A/B}(M)$, and that ΔN is minimal with respect to N and satisfies $N \oplus \Delta N : Rel(U \cup V, P \bowtie Q, F \cup G)$. Then $\delta put_\ell(M, \Delta N) = \delta put_{\ell'}(M, \Delta N)$.*

6 EVALUATION

We have implemented both naive and incremental relational lenses in Links. Given the lack of an existing implementation of relational lenses, for the naive version we implemented the lenses described by Bohannon et al. [2006a]. A benefit is a fairer performance comparison, as the non-incremental relational lenses are implemented using the same set operation implementation as the incremental version. We evaluate the performance of the optimised δput operations defined earlier.

Each performance experiment follows a similar pattern and are all executed on an Intel(R) Core(TM) i5-6500 with 16GB of RAM and a mechanical hard disk. The computer was running Ubuntu 17.10 and PostgreSQL 9.6 was used to host the database. Our customised Links version was compiled using OCaml 4.05.0. All generated tables contained a primary key index and no other indices.

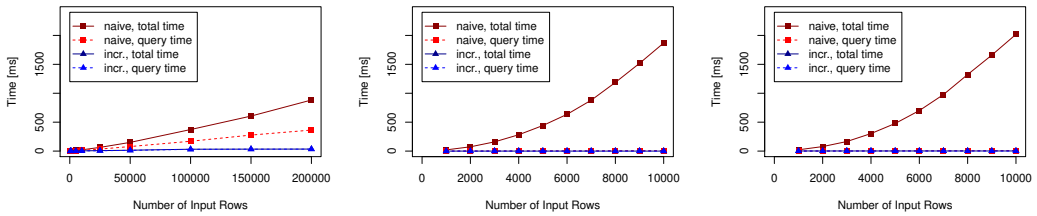
6.1 Microbenchmarks

6.1.1 Lens Primitives. We first evaluate lens change-propagation performance as a set of microbenchmarks over the lens primitives, using two metrics: total time to compute the source delta for a single lens given a view delta as input, or in the case of the naive lenses, to calculate new source tables, referred to as *total execution time*. We also measure the amount of the total execution time which can be attributed to query execution.

The following steps are taken for each benchmark:

(1) Generate the required tables with a specified set of columns and fill the tables with random data. For test purposes, the random data can either be sequential, a bounded random number or a random number. The microbenchmarks use the following tables:
   - Table t_1 with domain A, B and C and functional dependency $A \to B\ C$. Populated with n rows, with A calculated as a sequential value, B a random number up to $n/10$ and C a random number up to 100.
   - Table t_2 with domain B and D and functional dependency $B \to D$. Populated with $n/10$ rows with B being a sequential value and D a random number up to $n/10$.

(2) Generate lenses for the underlying tables and compose the required lenses on top of these. Lenses that use both t_1 and t_2 always start by joining the two tables on column B.



(a) select lens (b) projection lens (c) join lens

Fig. 11. Total and query execution time required by individual lens primitives vs. underlying table sizes

(3) Fetch the output view of the lens using *get* and then make changes in a systematic fashion as described for each setup. The changes are designed to affect a small portion of the database. Most changes are of the form: "take all rows with attribute A having some value, and update attribute B".
(4) Apply *put* for the first lens using the updated view. For the incremental lenses this means first calculating the delta from the view (not timed) and then timing δput which calculates the source delta. For naive lenses we measure the time required to recompute the source table using non-incremental *put*, but not the time needed to update the database. This process is repeated multiple times and the median value taken.

We repeat this process for each of the join, select and projection lenses. We exclude the time required to calculate the view delta or update the database because these operations only need to be performed once per lens, regardless of the number of intermediate steps defining the lens. Instead, we measure the performance of these one-time costs in §§ 6.1.2 and 6.1.3, and present a complete example involving two lens primitives in § 6.2.

Select example. The select benchmark uses both t_1 and t_2. We compose a select lens on top of the join lens with the predicate $C = 3$, which produces a view with an average size of $n/100$ entries. The view is modified so that all records with $0 \leq B \leq 100$ have their D value set to 5. This approach produces deltas of length 20 on average, containing one row removal and one row addition.

The naive *put* operation makes use of a single query and requires a total computation time of between $1ms$ and $867ms$ depending on the row count. While the performance is acceptable for small tables, it is still too slow for most applications as the tables become larger. It also shows how an unavoidable bottleneck is introduced, as the query time reaches up to $363ms$.

Depending on the row count, the incremental version only needs between $1ms$ and $40ms$ total computation time. Of this time, between $1ms$ and $39ms$ are used to perform the required queries, accounting for the majority of the total computation time. The execution time and query time scale proportionally to the data size. The incremental performance reflects the fact that the view is much smaller than the entire table, which needs to be recomputed for the non-incremental version. It may be possible to improve performance by configuring the database to index C, since this may reduce the query execution time. Indexing would not affect naive performance, since we always fetch the entire source tables.

Projection performance. We define a drop lens over the table t_1 removing attribute C, which is defined by attribute A with a default value of 1. The view is modified by setting B to 5 for all records where $60 < A < 80$. This process modifies 20 of the n records in the view.

The performance of the lens is shown in Figure 11b. As in the case of the join lens, the naive projection lens implementation quickly becomes infeasible, requiring a total execution time of over $2000ms$ as it processes 10000 records. The naive version spends up to $8ms$ querying the server. The incremental version is able to perform the δput operation in $1ms$ and less for the given row counts. This time includes the time required to query the database server for additional information.

Join example. The join benchmark uses the join lens defined over the two tables t_1 and t_2. We fetch the resulting view, which will contain n rows. After that we modify all records containing a value for B between 40 and 50 and set their C value to 5.

We benchmark the described setup with n values ranging from 1000 to 10000 in increments of 1000, timing the lens *put* duration for each n as specified. The performance results are shown in Figure 11c. The *put* operation for the naive join requires two queries but quickly becomes impractical. At 10000 rows it already requires over $2000ms$ and continues to rise quadratically. Of the computation time, approximately $1ms$ to $9ms$ depending on the table size is required for querying the database. While the query time taken by the naive approach is relatively low, this is due to the fact that the tables are relatively small and the time increases to hundreds of milliseconds as the table size grows to hundreds of thousands of rows.

In comparison, the incremental approach can scale to hundreds of thousands of rows and requires only $1ms$ to $2ms$ of both computation and query time for the given views. It requires 5 queries which are all simple to compute and return small views.

Summary. The above experiments show that incremental evaluation outperforms naive evaluation of relational lenses for data sizes up to 10K rows. We have also measured the performance of incremental evaluation for 200K rows to confirm that incremental performance continues to scale. Table 1 shows the number of queries, query evaluation time and total evaluation time for all three microbenchmarks discussed above.

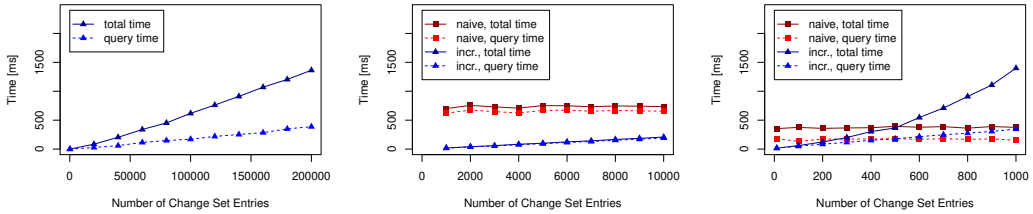
Table 1. Query counts and times for large data sizes

	select	project	join
query count	1	1	5
query $n = 200k$	$37ms$	$< 1ms$	$1ms$
total $n = 200k$	$39ms$	$< 1ms$	$2.5ms$

6.1.2 Delta Calculation Performance. While microbenchmarks on lens primitives give us some insight into the performance of the lenses, they do not account for the time required to calculate the initial delta, which is only required for incremental lenses. We modify the view of the join lens defined over t_1 and t_2 by fetching the view using *get*, and by then performing changes as done in the other experiments. Specifically we set B to 5 for all records where $0 < D < 10$. Given that this example does not have any selection lenses, the size of the view will always be n.

We measure the time taken to fetch the unchanged view from the database and then subtract it from a modified view. As in the previous examples we measure both the time required to query the database server as well as the total execution time on the client. We measure the time required for n values ranging between 100 and 200000. The results are shown in Figure 12a.

Both the query and execution time are approximately linear with respect to the number of input rows. We require between $< 1ms$ and $1360ms$ to compute the delta, of which just under half ($396ms$ for 200000 rows) is spent querying the database.



(a) Time needed to calculate delta between old and new view

(b) Time needed to apply source delta to the database

(c) Delta propagation time as a function of view update size (n=100000)

Fig. 12. Evaluation of delta calculation, delta application and delta propagation time as a function of view update size.

6.1.3 Delta Application Performance. We also measure the time it takes to apply a delta to a table. This process requires the generation of insert, update and delete SQL commands which must then be executed on the server.

We consider delta application for a single table. We use the table t_1 from § 6.1.1 and populate it with $n = 10000$. We generate a delta containing m entries, where a quarter of the entries produce $m/4$ insertions, another quarter produce $m/4$ deletions and the remaining half produce $m/2$ updates.

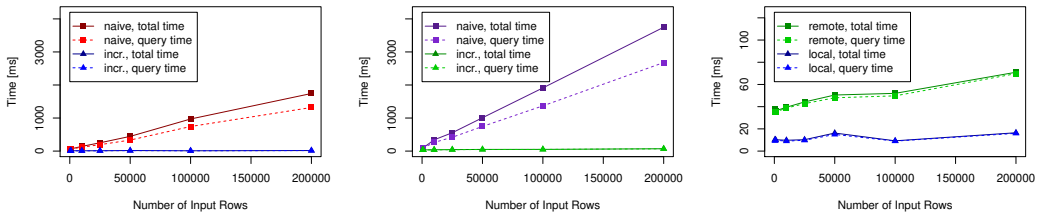
Given such a delta, we time how long it takes to produce the SQL commands from the already calculated delta with varying size m. The SQL update commands are concatenated and sent to the database together as a single transaction. As in the other cases we time both the total and query execution times, which are shown in Figure 12b. For the naive version, we generate an update command that deletes the current contents of the table and inserts the new contents. For the incremental version, we generate updates that insert, delete, or replace only affected records.

The figure shows that the naive version's performance is independent of the number of changes, requiring around $718ms$, most of which is spent querying the database. The incremental version, on the other hand, requires less time for the given change sizes and scales linearly, requiring between $18ms$ and $207ms$ depending on the delta size.

6.1.4 Varying Delta Size. In addition to varying the size of the underlying database tables we also consider how the size of the delta may affect the performance of an update. To do this we use the two tables t_1 and t_2 with $n = 100000$ and define a select lens on top of the join lens with the predicate $C = 3$. We then determine a b', starting from 0 and in steps of 100, so that modifying all records where $0 < B < b'$ by setting $D = 5$ produces a delta of size greater than m.

As in the other microbenchmarks we measure the total and query execution time taken to perform the δput of an already calculated view delta or, in the naive case, the time to recalculate the full source tables using put. We repeat this experiment for varying m values, ranging from 10 to 1000. The resulting execution times are plotted in Figure 12c.

As would be expected, the naive lens is relatively constant regardless of the number of entries in the delta. While the incremental version starts slows down as the delta becomes larger, eventually becoming even slower than the naive version, it does show that when the changes are sufficiently small the incremental method is much more efficient. When the size of the deltas exceeds around 500, however, it starts to become more efficient to recalculate the tables.



(a) Local database server (b) Remote database server (c) Local / remote performance

Fig. 13. The total/query execution time for the *put* operation applied to our DBLP database.

6.2 DBLP Example

In addition to the microbenchmarks we also perform some experiments on a real-world example involving the DBLP Computer Science Bibliography [Ley 2009], a comprehensive collection of bibliographic information about computer science publications. It is published as a large and freely available XML file with millions of records containing publications, conferences, journals, authors, websites and more.

Our example uses a table containing a collection of conference publications called *inproceedings* as well as a table of their respective authors *inproceedings_author*. This first table contains the *title* of the paper, the *year* it was published as well as the *proceedings* it is in, while the *inproceedings_author* table contains an entry for each author on every paper, allowing a single publication to have multiple authors. More information as well as example data is listed in the full version of this paper [Horn et al. 2018].

In order to determine how the application scales with varying database sizes, we generate the underlying tables by selecting a set of entries so that the join of *inproceedings* and *inproceedings_author* contains n rows. Given that set of entries, we select all entries in the complete *inproceedings_author* table, which have a corresponding entry in the subset of entries chosen.

Using these tables we join the two tables on the *inproceedings* attribute and then select all entries from PODS 2006. The Links code used to generate the lenses is shown below.

```
var joinL = lensjoin inproceedings_authorL with inproceedingsL on inproceedings;
var selectL = lensselect from joinL where proceedings == "conf/pods/2006";
```

As in the other examples we fetch the output of the select lens using *get* and then make a small modification. We then perform the *put* operation to apply those changes to the database. During the *put* we time the entire process of generating a delta from the view, calculating the delta for the underlying tables and updating the database for both the naive and incremental lenses. While timing we keep track of how much time was spent querying the database and the total time spent performing the operation. We also perform *put* using a database located on a remote server and compare it to a database located on the same machine.

Our performance results with the database hosted on the local machine are shown in Figure 13a. Similar to our earlier benchmarks, the incremental lenses perform favourably in comparison to the naive lenses. The naive lenses require linear time as the data grows and need up to 1743*ms* to update the database when the data grows to 200000 rows. A majority of the naive execution time (up to 1320*ms*) is used to query the entire database, and so optimising the local algorithms will have little effect on the overall performance.

The incremental lenses on the other hand perform much faster even as the data grows to hundreds of thousands of rows, requiring only between 9.5*ms* and 17.75*ms* total execution time depending on the size of the underlying tables. Of that time between 9*ms* and 17*ms* are used to query the database, making database performance the limiting factor for small datasets.

Figure 13b shows how the same application ran when using a database server situated on another computer. This affects the query time because the bandwidth is limited and the latency becomes higher. Less bandwidth means that the data loads more slowly, while higher latency imposes a delay per query. The total execution time of the naive version nearly doubles to up to 3748*ms*, of which 2679*ms* is used to query the database server, while the incremental version remains much faster requiring only up to 62*ms*, of which 59*ms* are used for querying the database.

Figure 13c directly compares the performance of incremental relational lenses when the database is located on the same or a different machine. Given that the bandwidth required is relatively small, the main additional overhead associated with using a remote database server is a roughly constant increase of about 30*ms*. Note that the data in Figure 13c is the same as that for incremental evaluation in Figures 13a and 13b, but with the scale of the y-axis adjusted to allow easier comparison of the two incremental versions.

7 RELATED WORK

Language-integrated query and web programming. Our approach is implemented in Links [Cooper et al. 2006], but should also be applicable to other functional languages with support for language-integrated query, such as Ur/Web [Chlipala 2015] and F# [Syme 2006], or the Database-Supported Haskell library [Ulrich and Grust 2015]. Our incremental relational lens definitions could also be implemented inside a database system. It would be interesting to explore what extensibility features could accommodate relational lenses in other languages.

Incremental computation. Incremental view maintenance is a well-studied topic in databases [Gupta and Mumick 1995]. We employ standard incrementalisation translations for relations (sets of tuples) [Qian and Wiederhold 1991; Griffin et al. 1997]. More recently, Koch [2010] developed an elegant framework for incremental query evaluation for bags (multisets of tuples), and Koch et al. [2016] extended this approach to *nested* relational queries. We think it would be very interesting to investigate (incremental) lenses over nested collections or multisets.

Incremental recomputation also has a large literature, including work on adaptive functional programming and self-adjusting computation [Acar et al. 2006; Hammer et al. 2014]. While closely related in spirit, this work focuses on a different class of problems, namely recomputing computationally expensive results when small changes are made to the inputs. In this setting, recording a large trace caching intermediate results can yield significant savings if the small changes to the input only lead to small changes in the trace. It is unclear that such an approach would be effective in our setting. In any case, to the best of our knowledge, this approach has not been used for database queries or view updates.

Our approach to incrementalisation does draw inspiration from the *incremental lambda calculus* of Cai et al. [2014]. They used Koch's incremental multiset operations in examples, but our set-valued relations and deltas also fit into their framework. Another relevant system, SQLCache [Scully and Chlipala 2017], shows the value of language support for caching: in SQLCache, query results (and derived data) are cached and when the database is updated, dynamic checking is used to avoid recomputing results if the query did not depend on the changed data. However, otherwise SQLCache recomputes the results from scratch. Language support for incremental query evaluation could be used to improve performance in this case.

Updatable views and lenses. Updatable views have been studied extensively in the database literature, and are supported (in very limited forms) in recent SQL standards and systems. We refer to Bohannon et al. [2006a] for discussion of earlier work on view updates and how relational lenses improve on it. Although updatable views (and their limitations) are well-understood, they are still finding applications in current research, for example for annotation propagation [Buneman et al. 2002] or to "explain" missing answers, via updates to the source data that would cause a missing answer to be produced [Herschel et al. 2017]. Date [2012] discusses current practice and proposes pragmatic approaches to view update. To the best of our knowledge, the work that comes closest to implementing relational lenses is Brul [Zan et al. 2016], which builds on top of BiGUL [Ko et al. 2016], a put-oriented language for programming bidirectional transformations. Brul includes the core relational lens primitives and these can be combined with other bidirectional transformations written in BiGUL. However, Zan et al. [2016] implement the state-based definitions of relational lenses over Haskell lists and do not evaluate their performance over large databases or consider efficient (incremental) techniques. They also do not consider functional dependencies or predicate constraints, so it is up to the programmer to ensure that these constraints are checked or maintained. Ko and Hu [2018] recently proposed a Hoare-style logic for reasoning about BiGUL programs in Agda, which could perhaps be extended to reason about relational lenses.

Object-relational mapping (ORM) is a popular technique for accessing and updating relational data from an object-oriented language. ORM can impose performance overhead but Bernstein et al. [2013] show that incremental query compilation is effective in this setting. We would like to investigate whether incremental relational lenses could be composed with more conventional (edit) lenses to provide ORM-like capabilities for functional languages.

Wang et al. [2011] considered incremental updates for efficient bidirectional programming over tree-shaped data structures, but not relations. There are several approaches to lenses that are based on translating changes, including *edit lenses* [Hofmann et al. 2012], *delta lenses* [Diskin et al. 2011], *c-lenses* [Johnson and Rosebrugh 2013] and *update lenses* [Ahman and Uustalu 2014]. None of these approaches has been applied to relational lenses as far as we know. Rather than utilize (and recapitulate) the needed technical background for these approaches, we have opted for concrete approach based on incrementalisation in the style of [Cai et al. 2014], but it would be interesting to understand the precise relationships among these various formalisms.

Bidirectional approaches to query languages for XML or graph data models have also been proposed [Hidaka et al. 2010; Liu et al. 2007]. However, to the best of our knowledge these approaches are not incremental and have not been evaluated on large amounts of data. There is also work on translating updates to *XML views* over relational data, for example Fegaras [2010]; however, this work does not allow joins in the underlying relations.

8 CONCLUSIONS

View update is a classical problem in databases, with applications to database programming, security, and data synchronisation. Updatable views seem particularly valuable in web programming settings, for bridging gaps between a normalised relational representation of application data and the representation the programmer actually wants to work with. Updatable views were an important source of inspiration for work on lenses in the functional programming languages community. There has been a great deal of research on lenses for functional programming since the influential work of Foster et al. [2007], but relatively little of this work has found application to the classical view update problem. The main exception to this has been Bohannon et al. [2006a], who defined well-behaved relational lenses based on a type system that tracks functional dependencies and predicate constraints in addition to the usual type constraints. Unlike updatable views in mainstream relational databases, relational lenses support complex view definitions (including joins) and offer

strong guarantees of correct round-tripping behavior. However, to the best of our knowledge relational lenses have never been implemented efficiently over actual databases.

In this paper we developed the first practical implementation of relational lenses, based on incrementalisation. Here, we again draw on parallel developments in the database and functional programming communities: incremental view maintenance is a classical topic in databases, and there has been a great deal of work in the programming language community on *adaptive* or *incremental* functional programming. We show how to embed relational lenses (and their associated type and constraint system) into Links and prove the correctness of incremental versions of the select, drop, and join relational lenses and their compositions. We also presented an implementation and evaluated its efficiency. In particular, we showed that the naive approach of shipping the whole source database to a client program, evaluating the put operation in-memory, and replacing the old source tables with their new versions is realistic only for trivial data sizes. We demonstrate scalability to databases with hundreds of thousands of rows; for reasonable view and delta sizes, our implementation takes milliseconds whereas the naive approach takes seconds.

Our work establishes for the first time the feasibility of relational lenses for solving classical view update problems in databases. Nevertheless, there may be room for improvement in various directions. We found a pragmatic solution that uses a small number of simple queries, but other strategies for calculating minimal deltas are possible. Developing additional incremental relational lens primitives or combinators, and combining relational lenses with conventional lenses, are two other possible future directions.

ACKNOWLEDGMENTS

Effort sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number FA8655-13-1-3006. The U.S. Government and University of Edinburgh are authorised to reproduce and distribute reprints for their purposes notwithstanding any copyright notation thereon. Perera was also supported by UK EPSRC project EP/K034413/1, and Horn and Cheney were supported by ERC Consolidator Grant Skye (grant number 682315). We are grateful to Jeremy Gibbons, James McKinna and the anonymous reviewers for comments.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison Wesley.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2006. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.* 28, 6 (2006), 990–1034. https://doi.org/10.1145/1186634

Danel Ahman and Tarmo Uustalu. 2014. Coalgebraic Update Lenses. *Electr. Notes Theor. Comput. Sci.* 308 (2014), 25–48. https://doi.org/10.1016/j.entcs.2014.10.003

François Bancilhon and Nicolas Spyratos. 1981. Update semantics of relational views. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 557–575.

Philip A. Bernstein, Marie Jacob, Jorge Pérez, Guillem Rull, and James F. Terwilliger. 2013. Incremental mapping compilation in an object-to-relational mapping system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013.* 1269–1280. https://doi.org/10.1145/2463676.2465294

Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. 2006a. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems.* ACM, 338–347.

Aaron Bohannon, Benjamin C Pierce, and Jeffrey A Vaughan. 2006b. *Relational lenses: a language for updatable views.* Technical Report MS-CIS-05-27. Department of Computer and Information Science, University of Pennsylvania.

Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. 2002. On Propagation of Deletions and Annotations Through Views. In *PODS.* 150–158.

Yufei Cai, Paolo G. Giarrusso, Tillmann Rendel, and Klaus Ostermann. 2014. A theory of changes for higher-order languages: incrementalizing λ-calculi by static differentiation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014.* 145–155. https://doi.org/10.1145/2594291.2594304

James Cheney, Sam Lindley, and Philip Wadler. 2013. A practical theory of language-integrated query. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming (ICFP '13).* ACM, New York, NY, USA, 403–416.

https://doi.org/10.1145/2500365.2500586

Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 153–165. https://doi.org/10.1145/2676726.2677004

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. 266–296. https://doi.org/10.1007/978-3-540-74792-5_12

George Copeland and David Maier. 1984. Making Smalltalk a database system. *SIGMOD Rec.* 14, 2 (1984).

C. J. Date. 2012. *View updating and relational theory.* O'Reilly.

Umeshwar Dayal and Philip A Bernstein. 1982. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems (TODS)* 7, 3 (1982), 381–416.

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10 (2011), 6: 1–25. https://doi.org/10.5381/jot.2011.10.1.a6

Leonidas Fegaras. 2010. Propagating updates through XML views using lineage tracing. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. 309–320. https://doi.org/10.1109/ICDE.2010.5447896

J Nathan Foster, Michael B Greenwald, Jonathan T Moore, Benjamin C Pierce, and Alan Schmitt. 2007. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 3 (2007), 17.

Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. 2010. Three Complementary Approaches to Bidirectional Programming. In *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*. 1–46. https://doi.org/10.1007/978-3-642-32202-0_1

Timothy Griffin, Leonid Libkin, and Howard Trickey. 1997. An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.* 9, 3 (1997), 508–511. https://doi.org/10.1109/69.599937

Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. http://sites.computer.org/debull/95JUN-CD.pdf

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 156–166. https://doi.org/10.1145/2594291.2594324

Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *VLDB J.* 26, 6 (2017), 881–906. https://doi.org/10.1007/s00778-017-0486-1

Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. 2010. Bidirectionalizing graph transformations. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 205–216. https://doi.org/10.1145/1863543.1863573

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2011. Symmetric Lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 371–384. https://doi.org/10.1145/1926385.1926428

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 495–508. https://doi.org/10.1145/2103656.2103715

R. Horn, R. Perera, and J. Cheney. 2018. Incremental Relational Lenses. *ArXiv e-prints* (July 2018). arXiv:cs.PL/1807.01948

Michael Johnson and Robert D. Rosebrugh. 2013. Delta Lenses and Opfibrations. *ECEASST* 57 (2013). http://journal.ub.tu-berlin.de/eceasst/article/view/875

Hsiang-Shang Ko and Zhenjiang Hu. 2018. An axiomatic basis for bidirectional programming. *PACMPL* 2, POPL (2018), 41:1–41:29. https://doi.org/10.1145/3158129

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM '16)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/2847538.2847544

Christoph Koch. 2010. Incremental query evaluation in a ring of databases. In *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 87–98.

Christoph Koch, Daniel Lupei, and Val Tannen. 2016. Incremental View Maintenance For Collection Programming. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 75–90. https://doi.org/10.1145/2902251.2902286

Michael Ley. 2009. DBLP: some lessons learned. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1493–1500.

Dongxi Liu, Zhenjiang Hu, and Masato Takeichi. 2007. Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007*. 21–30. https://doi.org/10.1145/1244381.1244386

Erik Meijer, Brian Beckman, and Gavin M. Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*.

Xiaolei Qian and Gio Wiederhold. 1991. Incremental Recomputation of Active Relational Expressions. *IEEE Trans. Knowl. Data Eng.* 3, 3 (1991), 337–341. https://doi.org/10.1109/69.91063

Raghu Ramakrishnan and Johannes Gehrke. 2003. *Database management systems (3. ed.).* McGraw-Hill.

Ziv Scully and Adam Chlipala. 2017. A program optimization for automatic database result caching. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* 271–284. http://dl.acm.org/citation.cfm?id=3009891

Perdita Stevens. 2007. A Landscape of Bidirectional Model Transformations. *GTTSE* 5235 (2007), 408–424.

Don Syme. 2006. Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. In *ML*.

Alexander Ulrich and Torsten Grust. 2015. The Flatter, the Better: Query Compilation Based on the Flattening Transformation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 1421–1426. https://doi.org/10.1145/2723372.2735359

Meng Wang, Jeremy Gibbons, and Nicolas Wu. 2011. Incremental updates for efficient bidirectional transformations. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011.* 392–403. https://doi.org/10.1145/2034773.2034825

Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views. In *Proceedings of the 5th International Workshop on Bidirectional Transformations, Bx 2016, co-located with The European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 8, 2016.* 77–89. http://ceur-ws.org/Vol-1571/paper_3.pdf